Consensus in Data Management: From Distributed Commit to Blockchain

Faisal Nawab[1] and Mohammad Sadoghi[2]

[1] *University of California, Irvine, USA; nawabf@uci.edu*
[2] *University of California, Davis, USA; msadoghi@ucdavis.edu*

ABSTRACT

The problem of distributed consensus has played a major role in the development of distributed data management systems. This includes the development of distributed atomic commit and replication protocols. In this monograph, we present foundations of consensus protocols and the ways they were utilized to solve distributed data management problems. Also, we discuss how distributed consensus contributes to the development of emerging blockchain systems. This includes an exploration of consensus protocols and their use in systems with malicious actors and arbitrary faults.

Our approach is to start with the basics of representative consensus protocols where we start from classic consensus protocols and show how they can be extended to support better performance, extended features, and/or adapt to different system models. Then, we show how consensus can be utilized as a tool in the development of distributed data management. For each data management problem, we start by showing a basic solution to the problem and highlighting its shortcomings that invites the utilization of consensus. Then,

Faisal Nawab and Mohammad Sadoghi (2023), "Consensus in Data Management: From Distributed Commit to Blockchain", Foundations and Trends® in Databases: Vol. 12, No. 4, pp 221–364. DOI: 10.1561/1900000075.

we demonstrate the integration of consensus to overcome these shortcomings and provide desired design features. We provide examples of each type of integration of consensus in distributed data management as well as an analysis of the integration and its implications.

1

Introduction

Consensus [49], [132]—which is the problem of making distributed nodes reach agreement—has influenced data management systems and research for many decades. This influence is due to consensus being a basic building block that can be used in more complex distributed data management systems while retaining correctness guarantees of the state of the data and its recovery.

Consensus becomes relevant to data management systems when data is distributed across multiple nodes. When multiple nodes are working together, many complexities arise due to communication uncertainties and the possibility of machine failures. This is the case in fundamental data management problems such as distributed atomic commitment and database replication [21], [56], [108], [129], [146]. Solving the intricacies of distributed coordination, network uncertainties, and failures in such complex data management problems is a daunting challenge. This has led many systems designers to utilize consensus as a tool to build more complex distributed protocols.

Consensus is solved in different ways depending on the system model and assumptions. One major factor in the design of consensus protocols is the failure model. The failure model can be a benign model—such as

crash fault-tolerance—where a node fails by stopping to engage in the protocol. Also, it can be a *byzantine* failure model [106], [132]—where a failed node can act in arbitrary ways including acting maliciously to influence the system negatively. In addition to the failure model, the network communication model also has an influence on the design and practicality of the proposed protocol. Communication models vary in a spectrum between a synchronous model—where time bounds on message reception are assumed—and an asynchronous model—where messages can be delayed indefinitely.

Variants of consensus algorithms are designed to answer unique challenges in different environments. Protocols that work best in a tightly-connected cluster might not be suitable for a distributed network separated by wide-area latency. Similarly, the workload plays an influence on whether to optimize for reaching consensus or learning about prior consensus outcomes. The goals of the protocol also play a part in how consensus algorithms are designed. Many protocols focus on achieving higher performance. However, some might optimize for lower latency while others optimize for higher throughput. Other than performance, a consensus algorithm might optimize for load balancing, faster recovery, or ease of understanding and implementation.

Consensus has renewed interest in the data management community in response to new problems. This interest started when consensus algorithms were utilized in replication and atomic commit protocols in distributed data management systems. With the growing interest in cloud computing in the 2000s, consensus has been explored as a means to design highly-available systems that are replicated across commodity machines. As cloud computing continued growing, consensus has also been explored in disaster recovery and multi-data center environments where data is copied and distributed across large geographic locations. More recently, cryptocurrency and blockchain-based applications reignited the interest in consensus and introduced a new breed of consensus algorithms that allow unique properties such as open membership to anonymous nodes [124], [155]. Data management systems has explored the use of such blockchain-based systems and consensus for applications spanning supply-chain management and decentralized finance, among others.

This monograph presents consensus as well as how it has been used to solve various distributed data management problems. The goal of this monograph is to provide a foundation for the reader to understand the landscape of using consensus protocols in data management systems as well as empower data management researchers and practitioners to pursue work that utilizes and innovates in consensus for their data management applications. This monograph is not meant to be a survey of consensus protocols nor it is a survey of data management systems that uses consensus. Rather, it presents the foundations of consensus and consensus in data management by presenting in more detail work that has been influential or representative of the data management areas we explore.

The monograph starts with a section to introduce the principles of consensus (Section 2). This section builds the foundation needed for the rest of the monograph to understand the consensus problem as well as the core consensus protocols that are widely-used in data management systems. Specifically, we will formally present the consensus problem and its guarantees as well as the space of system model and assumptions used by different protocols. Then, we present the paxos protocol in detail. Paxos [98], [99] is one of the most influential consensus algorithms that has been used—along with its variants—in many data management systems. We then present other consensus algorithms in different levels of detail to provide an intuition of the space of consensus algorithms including variants of the paxos protocol. Finally, we present how consensus is typically used in real systems using the abstraction of state-machine replication and what are other distributed systems problems that share properties with the consensus problem.

Section 3 presents background on the use of consensus in data management which provides an intuition of why and how consensus influences data management systems and the types of data management problems that invite the use of consensus protocols. This is done by providing a historical perspective of the development of distributed data management systems and how consensus has played a role in the various steps of this development. This section also presents background on data management systems that is needed for the rest of this monograph. It presents the system and data model of data management systems

that we utilize for the rest of the monograph. Also, it introduces the problems of transaction processing, concurrency control and recovery, as they are typically concerns that are involved while using consensus algorithms in distributed data management systems.

Section 4 presents how consensus is used for the distributed atomic commit problem, which is one of the most important problems in distributed data management systems. The section begins with an overview of the problem of atomic commitment and the significance of this problem in distributed and partitioned databases. This includes a detailed description of seminal protocols such as Two-Phase Commit (2PC) [9], [56], [108]. Then, we present more details about distributed atomic commit protocols that use consensus as a foundation. We present in more detail the paxos commit protocol [58] to represent a class of atomic commit protocols using consensus. We start from that description to discuss other atomic commit protocols that use consensus in different ways. We conclude the section with a discussion on the relation between the atomic commit and consensus problems. This relation stems from both protocols aiming to reach agreement across distributed nodes and show how many elements of atomic commit protocols and consensus protocols overlap and aim to provide similar properties.

Section 5 presents how consensus is used in replication protocols where data copies are distributed across different nodes. This section begins with an introduction to the problem of data replication and its significance in data management systems for performance and fault-tolerance. This includes presenting some early work on data replication and the ensuing concurrency control concerns. Then, we discuss how consensus can be used to solve the replication problem. In particular, we show how the state-machine replication abstraction has been used to enable multiple nodes to maintain copies of data that are consistent and recoverable. We also discuss how replication of individual participants in atomic commit protocols can be used as an alternative to the approaches we have shown in Section 4. We also present different variations of how consensus is used in different environments. In particular, we discuss the use of consensus in replicating for highly-available systems that gained popularity in cloud computing. Also, we present how consensus is adapted and used in environments that span large geographic locations such as multi-data center and geo-replicated systems.

Section 6 expands the scope of the crash-tolerant commit protocols to handle arbitrary failures. To this end, we explore in-depth the seminal fault-tolerant consensus protocol known as PBFT (Practical Byzantine Fault Tolerance) [34]. We present PBFT as the foundation for navigating and examining the consensus landscape. We further explore speculative, optimistic, linearized, and concurrent consensus designs. We conclude this section by examining the topology of consensus in the context of cross-shard and cross-chain designs. Our ultimate aim is to simplify and make the design of these intricate protocols accessible to a wide range of audiences, a stepping stone to further advancing this field.

Section 7 concludes the monograph with a summary and a discussion of future directions. We discuss the potential impact of utilizing and extending consensus in the areas of serverless computing, decentralized applications, and edge-cloud systems.

2

Principles of Consensus

At a high-level, consensus is the problem of reaching agreement on a single value among a set of distributed agents. Each agent may propose a value to be agreed upon. However, only one value out of the proposed values can be chosen. This means that any two agents cannot disagree on the chosen value. To avoid ambiguity with other types of values throughout the monograph, we refer to the value that nodes want to agree on as the *consensus value* or *c-value* for short.

In this section, we formalize this high-level definition of consensus and present examples of consensus algorithms. First, we begin by an overview of the system model that we will use throughout the monograph. This includes defining agents, failures, and communication as well as the variation in these models. Then, we present the problem of consensus formally using the system model. This includes the goals and guarantees of consensus as well as some basic theoretical results about the (im)possibility of consensus in different setups and system models. After that, we discuss how consensus solutions are typically used in distributed systems to provide more functionality beyond agreement. Finally, we conclude by a discussion of similar problems that are related to consensus or share some aspects with consensus.

2.1 System Model

2.1.1 Agents, State, and Events

The problem of consensus is applied to a distributed set of agents (also called nodes) $\mathcal{A} = \{a_0, \ldots, a_{n-1}\}$, where the number of agents is $n = |\mathcal{A}|$. Each pair of agents can communicate through a network link. However, any two agents do not share any memory or state. Therefore, communication is only possible via message passing.

Each agent a_i has an internal state s_j^i, where i is the agent's unique id and j denotes the number of state transitions that agent a_i took. An agent's state represents the memory of that agent that it uses to process requests and make decisions on state transitions. All agents start with a fixed initial state s_0. A state transition from s_j^i to s_{j+1}^i can be in reaction to an external or internal event. An example of an external event is a request from a client or a message sent from another agent. An example of an internal event is the expiration of a timer.

The set of events $\mathcal{E}$ is predefined for a consensus algorithm. Each event $e \in \mathcal{E}$ can have a set of parameters $e.p$, where each parameter $e.p[i]$ is identified by an attribute $e.p[i].attr$ and a value of that attribute $e.p[i].value$. For ease of exposition, we sometime refer to a parameter's value using the attribute name, hence $e.p[attr]$ would refer to the corresponding attribute value. For example, a message from one agent to another to propose a c-value is an event $e \in \mathcal{E}$ with parameters that may include the id of the agent that sent the request, $e.p[id]$, and the c-value the agent suggests, $e.p[c\text{-}value]$.

When an event is triggered at an agent, this leads to a state transition from the current state s_j^i to the next state s_{j+1}^i. This state transition can be deterministic or non-deterministic. Deterministic state transitions are ones where applying an event e to a state s_j^i would always lead to the same next state. Non-deterministic state transitions are ones where that is not the case—applying e to a state s_j^i may lead to transitioning to different states if applied in different times/conditions. An example of non-deterministic state transitions are ones that utilize random number generators or timestamps in the state transition.

2.1.2 Communication

Communication between agents is performed via network communication links where messages are sent from one agent to another. Networks may incur various unpredictable behavior such as delaying and reordering messages. Distributed algorithms typically define a model of communication to mask these complexities. Communication models are typically in the spectrum between a simple but less-realistic model of communication called *synchronous communication* and a more realistic model of communication called *asynchronous communication*.

The synchronous communication model assumes that a message that is sent by an agent will be received within some known time bound Δ. This means that if an agent sent a message at time t_1, that message will be received before time $t_1 + \Delta$. This model of communication simplifies real communication behavior where messages may be arbitrarily delayed or lost. However, it is a useful tool to study and design algorithms that are later adapted to more realistic communication models. When the communication model is synchronous, it is typical to assume that the system processing and timeliness is also synchronous. In this monograph, we assume that the time bound Δ captures both system processing and communication.

Distributed algorithms that build on the synchronous model typically adopt the following approach [106]. The progress of the system is modeled as moving in *pulses* (synchronous rounds), where a pulse is a time duration that corresponds to the time bound Δ in addition to an upper bound, Φ, on the time it takes an agent to process and send messages. In the beginning of the pulse, all agents are assumed to have received all messages sent in the previous pulse due to the known time bound Δ and processing bound Φ. Then, each agent experiences state transitions that correspond to the received messages (events). In response to these state transitions, an agent may generate messages that are sent to other agents within Φ time. These messages would be received and processed at the other agents by the time of the next pulse.

The asynchronous communication model is the more practical counterpart that factors in the possibility of arbitrarily delay and reordering of messages. In this model, a message that is sent from one agent to

another can be arbitrarily delayed. This arbitrary delay models various real network behaviors. It models the possibility of messages taking a longer time to be delivered due to network congestion and routing anomalies. It also models the possibility of message drops, since they can be considered as messages that are infinitely delayed. The challenge of using the asynchronous communication model when developing protocols is that it complicates the design and analysis of correctness and liveness.

The *partially synchronous* communication models [46] is a middle ground between the synchronous communication model—that does not reflect real network behaviors—and the asynchronous communication model—that introduces complexity to design and analysis. In a partially synchronous system, there are upper bounds on both communication, Δ, and processing, Φ. The difference compared to synchronous systems, is that in partially synchronous systems these bounds are not known. Another version of partially synchronous systems assumes known upper bounds for communication and processing, however, unlike the synchronous model, these bounds do not hold at all times. Instead, there is a *Global Stabilization Time* (GST), unknown to agents, where the upper bounds hold for some limited time after GST.

Throughout this monograph, algorithms and analysis will assume the use of the asynchronous model unless we mention otherwise. At times, the same algorithm may be applied to different communication models for different concerns. For example, analyzing the protocol correctness (safety) may assume an asynchronous model to prove that it is safe with a more realistic communication model, but may analyze the progress properties (liveness) by assuming a partially synchronous model due to the intractability of proving liveness in an asynchronous system where messages can be arbitrarily and indefinitely delayed.

2.1.3 Failure

An agent in the distributed system may experience failures that prevent its participation in the protocol. In this section, we present benign failure models and leave byzantine failures that may lead to arbitrary and malicious behavior of agents to Section 6.

An agent is considered to be *live* (not failed) if it performs the instructions of the protocol and continues to make state transitions in reaction to events it receives or triggers. In real deployments, agents may crash or become unresponsive. The following are ways to model such behavior.

The *fail-stop* failure model considers a failure of an agent that can happen at any time. A failed agent does not receive or send any messages and does not make any state transitions after it is failed. Also, once the agent fails, it cannot restart. The *omission* failure model considers a failure of an agent that can happen at any time. A failed agent may drop a subset of the messages that it sends to other nodes. Unlike fail-stop failures, an omission failure can be a temporary failure.

Consensus algorithms typically model failures by bounding the number of possible failures, f, that can be tolerated by the algorithm. This means that if the number of failures is up to f, then the algorithm would still achieve its correctness properties. Otherwise, some of the properties of the algorithm might not be satisfied. Some algorithms consider mixed types of failures, where there is a different f value for each type of failure considered.

2.1.4 Consensus Problem Statement

Consensus [49], [132] is the problem of ensuring that a group of agents $\mathcal{A}$ agree on a common value that we refer to in this monograph as the consensus value (c-value). Initially, agents are in an *undecided* state, where a c-value is not agreed upon. Some agents may *propose* c-values to the other agents. The agents engage in a consensus protocol to decide one of the proposed c-values to be the agreed upon value.

The concept of *deciding* (also called *choosing*) a c-value has a local and a global interpretation. The local interpretation considers a local view of a single agent. An agent (locally) decides a value when it marks a c-value as the agreed upon value. Once an agent decides a c-value, it commits to the decision and it does not decide another c-value. The global interpretation considers a logical view of the decided c-value as it pertains to the state of the whole system. When a c-value is decided in a system, it means that this is the value that all agents have (or will) agree on.

The concept of *proposing* a c-value refers to the ability of an agent to suggest a c-value to be decided. Multiple agents may propose different c-values. The consensus algorithm provides a method to decide one of these proposed c-values.

For a consensus algorithm to be correct, it needs to guarantee safety and liveness conditions. We describe safety and liveness conditions next.

Safety

Safety is the property to guarantee that an incorrect state is never reached. A safety condition is typically described in relation to the current state of the system that is comprised of the states of all its agents as well as all the transitions they have made to reach their current state.

Definition 2.1. (Consensus Safety Conditions) The following are the safety conditions for consensus:

- *Agreement:* Only one c-value may be decided.
- *Validity:* An agreed upon c-value must be one that has been proposed by an agent.

The agreement condition is satisfied if all agents in $\mathcal{A}$ agree on the same c-value. Consider that each agent $a_i \in \mathcal{A}$ has a state value called $a_i.c\text{-}value$ that represents what the agent believes to be the agreed upon c-value. A violation to the agreement condition happens if there exist two agents a_i and a_j that have different non-null values in their state's c-value, *i.e.*, $\exists_{a_i,a_j \in \mathcal{A}}(a_i.c\text{-}value \neq a_j.c\text{-}value) \wedge (a_i.c\text{-}value \neq null) \wedge (a_j.c\text{-}value \neq null)$. Note that a violation of the agreement condition can only happen if two agents disagree *after* both of them have decided a value. For example, if an agent a_i has decided a c-value while another agent a_j has not decided yet, this is not considered a violation of the agreement condition. The above formulation represents an undecided state by having a null value in c-value.

The validity condition is satisfied if the decided c-value is one that has been proposed by some agent prior to reaching agreement. Consider that a c-value, $\mathcal{C}$, has been decided in a collective state of all agents

$\mathcal{S}_i = \{s^0_{i_0} \dots s^n_{i_n}\}$, where $s^j_{i_j}$ is the state of a_j in $\mathcal{S}_i$. For validity to be satisfied, at least one agent a_j must have proposed the decided c-value $\mathcal{C}$ in a prior state. This means that there is a state s^j_l (where $l < i_j$) that has triggered proposing the c-value $\mathcal{C}$. If no such state exists, this is considered a violation of the validity condition.

Liveness

Liveness conditions are ones that describe guarantees on the progress of the system towards achieving the goal of the system. In the case of consensus, a liveness condition is a condition on guaranteeing that the system makes progress toward agreement and that eventually a c-value is decided.

Definition 2.2. (Consensus Liveness Condition) The following is the liveness condition for consensus:

- *Termination:* A non-faulty agent must eventually decide a c-value.

The termination condition is satisfied if all non-faulty agents in $\mathcal{A}$ can decide a c-value within a finite number of state transitions after a c-value is proposed. Specifically, for each agent a_i there is an associated number k_i that would represent the number of state transitions where it is guaranteed that a_i would have decided a c-value. This value, k_i, may be known or unknown. However, by proving that such a number is finite for all agents, the algorithm guarantees liveness. Sometimes, a liveness treatment takes an alternative approach by proving that a c-value is decided within a bounded time instead of a bounded number of state transitions. Parallels between the two approaches may be made.

Liveness of consensus algorithms varies from one algorithm to another and the possibility of achieving it varies according to the communication and failure models. An important result in this space is the following:

Definition 2.3. It is impossible to reach consensus in the presence of one fault in a system with an asynchronous communication model [49].

What this result means is that liveness cannot be guaranteed for the consensus problem where the communication model is asynchronous

and where the failure model allows the presence of at least one fault. The interested reader may learn about how this impossibility result is constructed from the original paper [49]. Here, we provide the sketch of that proof at a high-level. The proof starts by considering a system of agents with collective state $\mathcal{S}$. The initial state, $\mathcal{S}_0$, is an undecided state (a c-value has not been decided yet), since agents have not coordinated yet. Events applied to $\mathcal{S}_0$ will lead to a state transition to $\mathcal{S}_1$ that would either be decided or undecided. Likewise, at any state $\mathcal{S}_i$, there are events that may lead to a state transition to a decided or undecided state $\mathcal{S}_{i+1}$. The proof shows that this is the case for any $\mathcal{S}_i$, which means that there is the possibility of taking an infinite sequence of state transitions that are all undecided. To show that this is the case, the proof includes lemmas to show that any undecided state $\mathcal{S}_i$ would have transitioned to an undecided state. Also, it shows that an inopportune failure or message reordering due to the asynchronous communication model can always lead to the next event triggering a transition to an undecided state.

Despite being impossible in asynchronous systems, liveness can be guaranteed and reasoned about by relaxing the synchrony model [44], [46]. Also, the impossibility of liveness does not mean that safety cannot be guaranteed. There are consensus protocols that are safe despite assuming an asynchronous communication model and a failure model where nodes may fail.

Learning

In addition to reaching consensus, agents or clients may want to learn the decided c-value. This is called the *learning* process.

Definition 2.4. (Learning Safety) a learning process is safe if it only learns a c-value if it is decided.

This condition is satisfied if the agent that learns the c-value—or responds to a client with the c-value—must be in a state where the c-value is decided. Consider that the agent a_i is in state s^i_j which is part of the collective state $\mathcal{S}_k$ when it learns or responds to a learner client. The c-value must have been decided in a collective state $\mathcal{S}_l$ where $l < k$.

2.2 Consensus Algorithms

The significance of the consensus problem has led to many consensus solutions. In this section, we provide a detailed description of paxos [98], [99], which is a widely-used consensus algorithm and the inspiration and basis for many consensus algorithms. We conclude this section with a brief survey of consensus protocols and variants that aim to solve various problems in different system models and environments.

2.2.1 Paxos

Paxos is a consensus protocol that is designed for an asynchronous communication model. The number of agents in paxos is $n = 2f + 1$, where f is the number of tolerated failures. Paxos divides the logical roles to (Figure 2.1): (1) *proposers* that attempt to propose a new c-value to be decided. There need to be at least $f + 1$ proposers that are typically colocated with the agents. (2) *acceptors* that are used to maintain the state of consensus. Acceptors receive proposed c-values from proposers and decide whether to accept them or reject them. There are $2f + 1$ acceptors, each colocated with an agent. Acceptors do not communicate with each other. Instead, they receive requests from proposers that aim to change the state of enough acceptors for consensus to be achieved. (3) *learners* that are used to learn the decided c-value by asking acceptors.

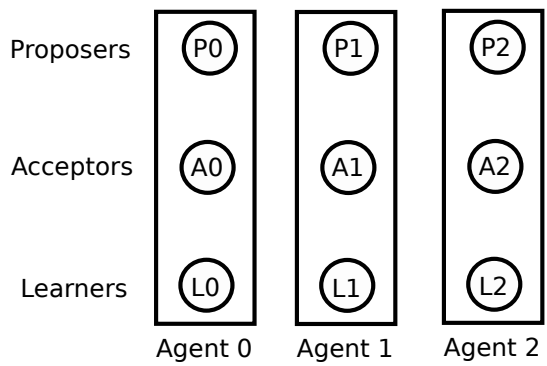


Figure 2.1: Logical representation and organization of paxos components.

For clarity of exposition, in this section, we assume that there are $2f+1$ acceptors colocated with $2f+1$ proposers. This means that each agent out of the $2f+1$ agents has a single proposer and a single acceptor (Figure 2.1). This is typically how paxos is deployed in practice.

Design Principles

Paxos combines three design principles to reach consensus. The first is the principle of *rounds*, where the protocol proceeds in numbered epochs. The number of the epoch is called a round number or *ballot number*. When a proposer receives a message from a proposer, it processes it only if it has a ballot number that is larger or equal to every other ballot number it has seen before. This allows proposers to break deadlocks in a round by advancing to a round with a larger ballot number.

The second principle is *majority voting*. Every step of achieving agreement in the paxos protocol involves the proposer sending a request to a majority of nodes. If a majority of nodes agree on that step, then the step is successful. Otherwise, the step has failed. The significance of requiring a majority for the success of a step is that any two majority quorums intersect. Therefore, if two proposers have made conflicting steps, then that conflict is guaranteed to be detected because the two majority quorums would necessarily intersect.

The third principle is that paxos is a *leader-based protocol*. For a proposer to propose a c-value, it first needs to become a leader. Only after becoming a leader would a proposer be able to propose a c-value. This is done by dividing the operation of a proposer into two phases. The first phase is the leader election phase (Phase 1), where the proposer asks a majority of acceptors if it can be the leader of a round. If the leader election is successful (a majority of acceptors agree), then the new leader can proceed to the replication phase (Phase 2). In the replication phase, the proposer asks a majority of acceptores if they accept a c-value that it proposes. If a majority agree, then the c-value is decided. Otherwise, it is not.

Agents State

Each agent a_i contains a proposer, $proposer_i$, and an acceptor, $acceptor_i$. The state of the proposer, $s_j^{i,pr}$, contains a ballot number, $s_j^{i,pr}.ballot$, that represents the round number that this proposer is currently in. Each proposer uses unique ballot numbers that are not used by other proposers. This is typically implemented by making the ballot number be a lexicographically ordered integer pair (r,id), where r is a monotonically increasing integer and id is a unique identifier of the proposer. A proposer can update the ballot number as long as the new value is larger than the current value.

The state of the acceptor, $s_j^{i,ac}$, contains the following: (1) the highest ballot number it has seen, $s_j^{i,ac}.highest\text{-}ballot$. If a received message belongs to a round with a smaller ballot number, the acceptor ignores that message. This ballot number is initially set to a number that is smaller than any possible proposer ballot number. (2) The c-value that the acceptor has already accepted, $s_j^{i,ac}.accepted$, if any. If the acceptor accepted more than one proposal, then only the one with the highest ballot number is recorded. This is used for the acceptor to record whether it has accepted a proposed c-value from a proposer. This includes both the c-value of that accepted proposal, $s_j^{i,ac}.accepted[c\text{-}value]$, and its corresponding ballot number, $s_j^{i,ac}.accepted[ballot]$. This state variable is initially empty.

Algorithms

The paxos protocol is driven by the proposers committing values on behalf of clients. When a proposer wishes to propose a new c-value, it needs to go through two phases: the leader election phase (sometimes referred to as Phase 1), and the replication phase (sometimes referred to as Phase 2). Figure 2.2 provides a schematic representation that will be used in the following description.

Leader Election (Phase 1). The purpose of the leader election phase is to prepare for the replication phase. This preparation aims to perform two tasks: (1) prevent proposers with smaller ballot numbers from proposing a c-value, and (2) discover any previous c-values that were potentially decided.

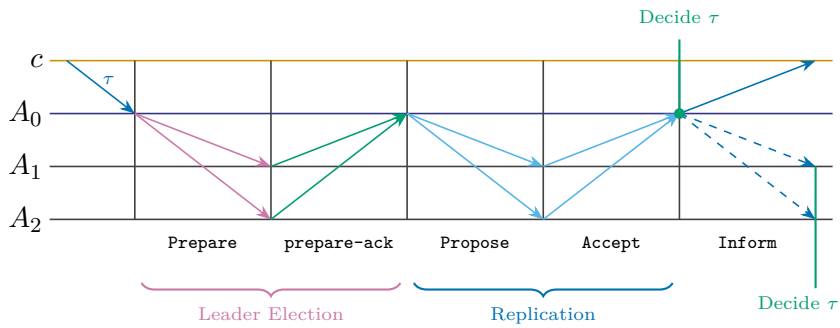


Figure 2.2: A schematic representation of the normal-case operation of deciding a c-value using the paxos protocol.

The proposer achieves both tasks by the following message exchange. The proposer sends a prepare message (also called Phase1A message) to at least a majority of acceptors. This prepare message contains the proposer's ballot number. When an acceptor a_i receives a prepare message, it checks the associated ballot number, *prepare.ballot* and compares it with the highest ballot it has seen so far, $s_j^{i,ac}.highest\text{-}ballot$. if the received ballot is equal or higher than what is previously seen ($prepare.ballot \geq s_j^{i,ac}.highest\text{-}ballot$), then the acceptor updates its highest-ballot state variable to *prepare.ballot*. Then, it responds with a prepar-ack message (also called a Phase1B message). Otherwise, if $prepare.ballot < s_j^{i,ac}.highest\text{-}ballot$, then the acceptor discards the prepare message. The acceptor may choose to send a negative acknowledgment to avoid making the proposer wait for a long time.

The prepare-ack message contains three parameters. The first is the ballot number from *prepare.ballot* denoted *prepare-ack.ballot*. The second and third parameters correspond to the largest c-value previously accepted by the acceptor, stored in the state variable $s_j^{i,ac}.accepted$. One parameter ($prepare\text{-}ack.accepted\text{-}cvalue$) is of the accepted c-value read from $s_j^{i,ac}.accepted[c\text{-}value]$. The other ($prepare\text{-}ack.accepted\text{-}ballot$) is of the ballot number associated with that accepted value, $s_j^{i,ac}.accepted[ballot]$. These state variables are set if the acceptor has ever received a replication phase propose message previously from a previous leader, as

we will explain shortly. If the acceptor did not receive such a message previously, then the two corresponding parameters are set to null in the prepare-ack message back to the proposer.

When a proposer receives a prepare-ack message, it first checks whether the message includes information about previously accepted proposals. If it does, then the proposer checks whether the previous proposal's ballot number (*prepare-ack.accepted-ballot*) is the highest it has received so far in this round. If it is, then the proposer maintains both the corresponding c-value and ballot number to be used in the replication phase. The proposer maintains this information in the state variables *proposer.accepted*[*c-value*] and *proposer.accepted*[*ballot*].

The proposer waits until it hears a majority of prepare-ack messages. Once a majority of messages are received, the proposer considers itself a leader and is ready to move to the next phase—the replication phase—where it can propose a c-value. If a majority response is not received within a predefined time threshold, the proposer considers that its attempt to become a leader failed—potentially due to enough acceptors having observed higher ballot numbers. The proposer increases the ballot number and has the option of retrying.

When a proposer successfully receives a majority of prepare-ack messages in a round, it is in the *leader state*. The implications of proposer, a_i^p, being in the leader state in round b (where b is the corresponding ballot number) are the following: (1) No other proposer can successfully become a leader or propose a c-value if its ballot number is less than b. This is because a majority of acceptors responded with prepare-ack for b, which is in part a promise not to respond to any message with smaller ballot numbers. This enables the new proposer to know that an old leader cannot appear and propose a c-value after a_i^p became a leader. (2) If a c-value has been decided by a prior leader (that has a ballot number q less than b), then it is guaranteed that a_i^p would have learned about this value through the received prepare-ack messages. This is because a previously decided value (with ballot number q) is necessarily accepted by a majority of acceptors (as we will see in the replication phase.) Let's call that quorum that accepted the value at q as $\mathcal{A}_q$ and call the majority quorum that responded to the leader in round b as $\mathcal{A}_b$. There is at least one acceptor a_i^a that is in the intersection

of $\mathcal{A}_q$ and $\mathcal{A}_b$. Since a_i^a accepted the value in q, it would include that accepted value in its response to the proposer in b.

Replication (Phase 2). The purpose of the replication phase is to propose a c-value to be decided by acceptors. This is performed by a message exchange between a leader (proposer that successfully performed leader election) and a majority of acceptors. A successful replication phase leads to deciding a c-value.

A leader starts the replication phase by picking a c-value to propose. This is done by observing the responses that were received in the leader election phase. Specifically, the leader observes whether there were reports of previously accepted proposals from prepare-ack. If there are more than one, the leader picks the one with the highest corresponding ballot number. This can be read from *proposer.accepted*[*c-value*] and *proposer.accepted*[*ballot*] that we described in the leader election phase. This c-value is picked by the leader to be the one that it will try to propose in the replication phase, even if it was proposed by another leader previously. The reason why this is done is because this previously accepted c-value might have been decided. If no previously accepted values were reported in the leader election phase, then the leader has the opportunity to pick its own c-value to propose.

Once a c-value is picked, the leader sends a propose message (also called Phase2A message) to a majority of acceptors. The propose message consists of a ballot number (that corresponds to the ballot number used by the leader in the prepare message) and a c-value to propose (that was picked after observing the prepare-ack responses from acceptors.)

When an acceptor receives a propose message, it checks its ballot number. If it is higher or equal to the highest ballot number it has ever seen, then the proposer accepts the proposal. Otherwise, the propose message is discarded. When an acceptor accepts a proposal it performs two actions. First, it updates the highest accepted proposal state variable, $s_j^{i,ac}.accepted$, to the received proposal and its ballot number. Second, it responds with an accept message (also called a Phase2B message.)

The leader waits until it hears from a majority of acceptors. Once a majority of accept messages are received, the leader considers the c-value to be decided. Some implementations may send a special inform message to notify acceptors of the decided c-value.

Correctness

We provide here a high-level sketch of the correctness of paxos and refer to other sources for detailed and more formal proofs [98], [99]. We focus in this part on the two safety properties defined in Definition 2.1. Validity is more straightforward to show than agreement. Validity—a guarantee that a decided c-value is one that has been proposed by an agent—is guaranteed because any decided c-value is one that has been proposed by a leader in the replication phase. The leader either proposes its own c-value or a previously accepted c-value that is proposed by another agent. The rest of this section focuses on the other safety guarantee—agreement.

The correctness of paxos stems from the intersection between its majority quorums. Specifically, if a c-value, c, is decided, then this means that a leader l has successfully received at least a majority of accept messages. Assume that the leader decided this value in round b. Paxos ensures that no other leader, l', can successfully decide a c-value other than c.

To show this, assume to the contrary that another value c' is decided. There are two cases of l' deciding a c-value c' in another round q (we omit the trivial case where $b = q$):

1. $q < b$ (Figure 2.3): assume that the quorum Q_q^r is the quorum of acceptors that accepted the the c-value c' in q. Also, assume that Q_b^{le} is the leader election quorum that responds with prepare-ack messages in round b. There exists at least one agent A in the intersection of Q_q^r and Q_b^{le} since they are both majority quorums. Since q is less than b, this means that A accepted c' in q before responding with a prepare-ack in b. This means that by the time it is responding with a prepare-ack in b, it has already accepted c', which means that it will include it in the prepare-ack message. This leads to l picking the value c' as its proposal in the propose message in round b. In this case, the proposal at b is the same as the proposal at q ($c = c'$), which is a contradiction to our assumption that they are different values.

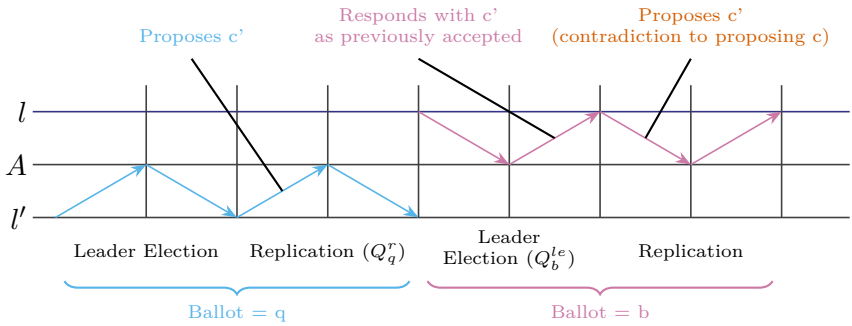


Figure 2.3: A schematic representation of an example of case 1 (q<b) in the proof sketch of paxos correctness.

2. $b < q$: consider an agent A in the intersection of the quorum Q_b^r that accepted the c-value c in b and the quorum Q_q^{le} that responded with prepare-ack messages to l' in q. This case is further divided into two sub-cases:

   a. The prepare message from l' is received at A before the propose message from l is received (Figure 2.4). In this case, A would not accept the proposal from l because it has already seen a message with a higher ballot number. This is a contradiction since c is not decided.

   b. the prepare message from l' is received at A after the propose message from l is received. In this case, A would respond with the c-value c that it accepted as part of the prepare-ack to l'. Similar to case 1 above (Figure 2.3), this leads to l' using the c-value c as its proposal, and thus $c = c'$, which is a contradiction.

Paxos Liveness

The paxos protocol does not guarantee liveness due to the impossibility result mentioned in Definition 2.3 [49]. Such impossibility of achieving liveness can be manifested in different scenarios. One example of such scenario is the following (Figure 2.5). Consider a paxos instance with

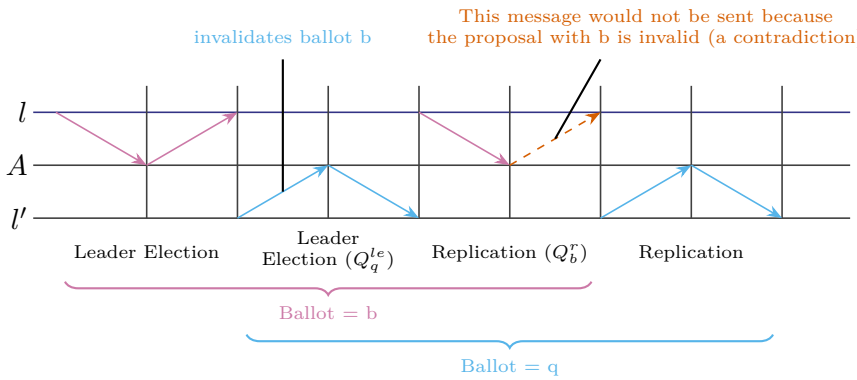


Figure 2.4: A schematic representation of an example of case 2(a) (q>b and the prepare from l' is received at A before the propose from l) in the proof sketch of paxos correctness.

three agents, A, B, and C. Initially, agent A attempts to become a leader, so it sends a prepare message to B with ballot number 1. Agent B responds with a prepare-ack since this is the highest ballot it has seen so far. Agent A receives the prepare-ack and becomes the leader of round 1. However, in the meantime, agent C also attempts to becomes a leader and sends a prepare message to B with ballot number 3. Agent B responds with a prepare-ack to C since 3 is the highest ballot number it has seen so far. Now, when agent A sends a propose message with ballot number 1, it will not be accepted since it is not the highest ballot seen by B. At this moment, agent A may want to retry with a higher ballot number, 4, and sends a prepare message to B. Agent B can now accept this new prepare, which invalidates any future propose messages by C with ballot number 3.

Observe in the previous scenario how two aspiring leaders, A and C, may continuously invalidate each others' proposals. This scenario can continue indefinitely, which demonstrates how the message communication and delivery patterns may lead to a paxos scenario that cannot reach agreement. This is possible because the communication and delivery patterns are arbitrary due to the asynchronous communication model (Section 2.1.2). In the asynchronous communication model, messages can be delayed indefinitely, which may cause two agents to

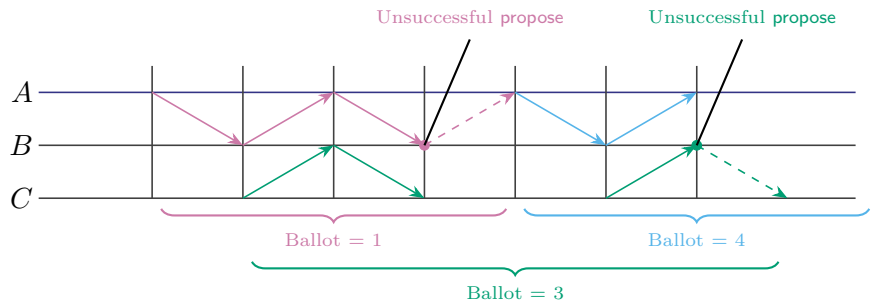


Figure 2.5: A schematic representation of a paxos livelock scenario with three nodes. The color of the arrows represents their corresponding ballot numbers.

continuously invalidate each other without learning about the other agent's rounds.

Relaxing the communication model is one way to reason about liveness. For example, assuming a synchronous or partially synchronous communication model enables us to assume that communication latency between agents is bounded (Section 2.1.2). With such bounds, mechanisms to guarantee liveness were proposed [44], [46]. In these approaches, the guarantee of receiving messages can be used to design a protocol that is guaranteed to terminate after a certain number of rounds. This is possible because messages sent at the beginning of a communication round are guaranteed to be delivered—unlike the paxos livelock scenario where two aspiring leaders do not receive each others messages. In a real system, these approaches guarantee liveness as long as the assumptions about the weaker communication model are met. An alternative to weakening the communication model is to weaken the consensus problem by introducing initial assumptions about the decided values and/or to tolerate approximate or probabilistic agreement [15], [28], [45]. Similarly, other methods place restrictions on the system model. This includes the use of failure detectors, which are separate components that report—potentially inaccurately—on suspected failures but these reports are eventually corrected. It is shown that with the assumption that a failure detector exists, it is possible to build consensus protocols that guarantee liveness in an asynchronous model [36]. However, the

existence of such failure detectors remains restricted and assumes that these failure detectors communicate using a synchronous or partially synchronous links.

Learning

In addition to proposers and acceptors, learners are components that aim to learn the decided value by communicating with acceptors. A learner learns that a c-value is chosen if a majority of acceptors have accepted it. This can be achieved in different ways. One way is for the learner to poll acceptors and ask for the c-values they accepted. If a majority of acceptors accepted the same c-value, then the learner considers that c-value as the decided one. Another algorithm is to have acceptors send messages proactively to learners whenever they accept a c-value. The learner knows a c-value is decided once a majority of acceptors have sent the same c-value.

The learning algorithm can also be optimized by having proposers, acceptors, and learners cache the decided c-value and propagate it to other agents. For example, a proposer that decides a c-value may send a special inform message to acceptors and learners notifying them of the outcome. Anyone that has received such a message can respond immediately with the decided c-value for future inquiries instead of repeatedly polling a majority of acceptors.

Multi-Paxos

A common optimization that is used when paxos is deployed in practice is the multi-paxos optimization [35], [148]. The goal of multi-paxos is to reduce the needed number of message exchanges. Paxos commits a new c-value by proceeding through two phases: a leader election and replication phases. An observation about these two phases is that in the leader election phase, the proposer aspiring to become a leader, does not need to know what c-value it wants to commit yet. Therefore, it is possible to perform the leader election phase early before the c-value is ready to propose. This allows removing the leader election communication exchanges from the path of execution of committing new c-values.

This multi-paxos optimization is particularly useful in practical scenarios where consensus is used continuously to decide c-values (for example, we will present later how using paxos in state machine replication entails using paxos to commit a c-value for each log position in a replicated log.) Therefore, instead of having to do two phases of communication to commit each c-value in the path of execution, the leader election is performed early, and then the replication phase is performed when the c-value is known and ready to be committed.

A further optimization in multi-paxos reduces the amount of messages that need to be sent for leader election. Even if leader election is performed early out of the path of execution, its corresponding communication affects bandwidth. Also, continuously performing early leader election of future commitments might interfere with the replication phases of ongoing commitments. To this end, multi-paxos *consolidates* leader election messages. Specifically, instead of sending individual prepare messages for a group of paxos instances, a single prepare message is sent with the range of identifiers for the future paxos instances.

For example, consider that future instances are numbered in chronological order, where the first paxos instance has id 0, the second has id 1, and so on (this is similar to the use case in state machine replication we will present later.) A proposer may send a prepare message indicating in it that it wants to be a leader for instance ids i to j. This represents a small overhead increase compared to the original prepare message. When an acceptor receives this message, it applies the prepare logic of all the paxos instances in the range, and responds only if the conditions apply to all relevant instances. Like the consolidated prepare messages, the acceptor consolidates the prepare-ack messages.

It is possible to set the range of paxos instances to be to infinity, meaning that a proposer becomes the leader of all instances greater than a specified paxos instance number. This requires both the proposer and acceptor to have efficient representations of paxos instances and their state that enables expressing the state of an infinite range of paxos instances.

2.2.2 Other Consensus Protocols

Due to the importance of the consensus problem, there has been a lot of activity and work to both develop new consensus protocols and extend existing consensus protocols.

Paxos has been an influential consensus protocol in terms of derivative work and variants of the paxos protocol. Follow up works by Leslie Lamport explore generalizations of the paxos protocol. This includes Fast Paxos [101]—that introduces the concept of a fast path to decide c-values using a single round of communication—and Generalized Paxos [100]—that extends paxos to enable reaching agreement on partial order rather than total order. A body of work extends paxos to manage reconfiguration [55], [102]–[105], [112]—which is the problem of changing the set of agents running the consensus protocol. Load balancing is an important extension to paxos to avoid the overhead that is placed on the leader. This led to extensions such as S-Paxos [23] that distributed some of the work of the leader to other replicas which reduces the load on the leader. Works such as DPaxos [126] and WPaxos [8] explore extensions such as hierarchy, locality, and sharding to enable better performance in geo-replication settings. Utilizing variants of paxos such as Fast Paxos [101] and Generalized Paxos [100] has also been explored to reduce the amount of wide-area communication in protocols such as MDCC [95]. Mencius [116] is a multi-leader system that is based on paxos. It aims to enable faster latency by partitioning log entries across agents and serving client requests using the closest agent and its assigned log entries. Moraru *et al.* [122] propose Egalitarian Paxos (EPaxos) that aims to reduce communication complexity by utilizing a fast path design and information about conflicts. Paxos variants to utilize the special properties of the network infrastructure include Ring Paxos [117], Sift [87], and NOPaxos [110]. Also, work to avoid the overheads and bottlenecks of networking is proposed, such as PigPaxos [38] that studies aggregation and piggybacking for paxos, and work that studies pipelining and batching for paxos [138]. Whittaker *et al.* studies the modularity of consensus protocols and proposed paxos variants that emphasize this modularity [151]–[154].

Paxos relies on the intersection of majority quorums to ensure safety. However, it has been shown that majority quorums are not necessary for safety. Instead, Howard *et al.* [82] made the observation that what is necessary for safety is only the intersection of leader election and replication quorums.[1] Therefore, it is not necessary to have different quorums from the same phase intersect with each other (e.g., two leader election quorums do not need to intersect).

This makes quorum definitions more flexible, allowing different optimizations as we will show in different places throughout the monograph. An example of quorum definitions that are allowed is shown in Figure 2.6. The first is grid quorums, where acceptors are organized in a $g \times g$ grid, where $g = \sqrt{n}$. A leader election quorum can be any column in the grid, and a replication quorum can be any row in the grid. This reduces the size of quorums from majority quorums ($\lfloor \frac{n+1}{2} \rfloor$) to $\sqrt{n}$. Fast Flexible Paxos [81] extends on flexible paxos [82] to study quorum intersection and how to reduce the intersection requirements.

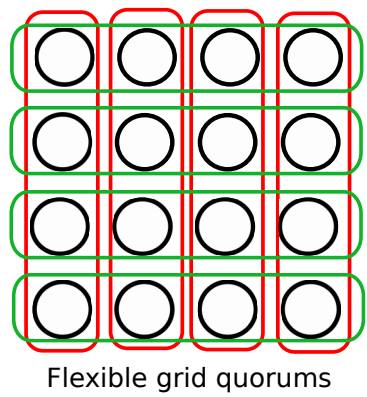


Figure 2.6: An example of a flexible grid quorums definition where circles represent agents, red rectangles represent leader election quorums, and green rectangles represent replication quorums.

Other than paxos, there has been a number of consensus protocols. Viewstamp Replication [127] is a primary-based replication system that in normal-case operation acts similarly to multi-paxos. Viewstamp

[1] Observe how the intersection conditions in the correctness section above were all between a leader election and replication quorums.

Replication utilizes a special view change process to tolerate failures. RAFT [128] is a consensus-based replication system that aims to provide a more understandable solution for consensus. Zab [85] is an atomic broadcast algorithm that is used to propagate state from a primary to a set of backup nodes in a consistent and fault-tolerant manner. These various protocols have many common features but also differ in various ways. Vive La Difference [149] studies these similarities and differences.

The atomic broadcast problem is closely related to the problem of consensus [71]. At a high level, the atomic broadcast problem is to ensure that all messages sent by agents are totally ordered. In the next section, we discuss how consensus is typically used to achieve such total ordering in replicated logs.

2.3 Using Consensus

Consensus has been widely-used in distributed systems to enable coordination and replicating state across devices. In this section, we present the common ways of using consensus for distributed coordination as well as common optimizations.

State Machine Replication

One of the most common ways of using consensus in distributed systems is as a basic building block for state-machine replication (SMR) [97], [109], [139]. SMR is a method that enables distributed agents to synchronize their state. The goal is to have the state of any pair of agents to be identical without the aid of any centralized process or storage.

The *State Machine* is the main abstraction for synchronization that the different agents try to synchronize. This State Machine, for example, can be a representation of the state of the file system, database, etc. The set of possible states for a State Machine is $\mathcal{S}$. The State Machine accepts commands from the set of possible commands $\mathcal{C}$. Applying a command $c \in \mathcal{C}$ to a state $s \in \mathcal{S}$ leads to a deterministic state transition to state s'. The deterministic nature of the transition means that any agent would make the same state transition if it starts with the same state and applies the same command.

For example, a SMR system for a file system may start with a state of a folder that contains three files. If a command is applied to that state to add a new file, then the state transitions to a folder with four files. This step can be made to be deterministic by ensuring that running the command will always lead to the same outcome given the original state. A non-deterministic step can be one that depends on external factors such as a random number generator or the wall clock time.

A common approach for SMR is to order all the received commands in a replicated log, that we will call the *SMR log*. Each agent has a copy of the SMR log and they coordinate to ensure that the i^{th} entry in any SMR log is identical to the i^{th} entry in other SMR logs. Each agent starts from a genesis state, s_0, for the State Machine that all agents share. Then, an agent processes commands according to their order in the SMR log. This ensures that all agents start from the same state and apply commands in the same order. Because commands lead to a deterministic state transition, the state of all agents is identical after applying the same number of commands. An example of SMR is shown in Figure 2.7.

Consensus for State Machine Replication

Consensus protocols are typically used to solve the SMR problem. Specifically, to build a SMR system, there needs to be a mechanism to achieve agreement on the state of the replicated SMR log. Consensus can help achieve that agreement. However, consensus is typically formulated as the problem of achieving agreement on the state of a single value. In the case of SMR, we need to achieve agreement on the state of a growing SMR log.

Consensus protocols are applied to the SMR problem by considering each position in the SMR log as an independent consensus instance. Basically, we are reducing the problem of achieving agreement on the state of the whole SMR log to achieving agreement on each log position individually. This reduces the complexity of the problem and invites a direct application of consensus solutions to the SMR problem.

When a command is received by an agent, the agent picks the next available SMR log position, L_i, and engages in a consensus process with

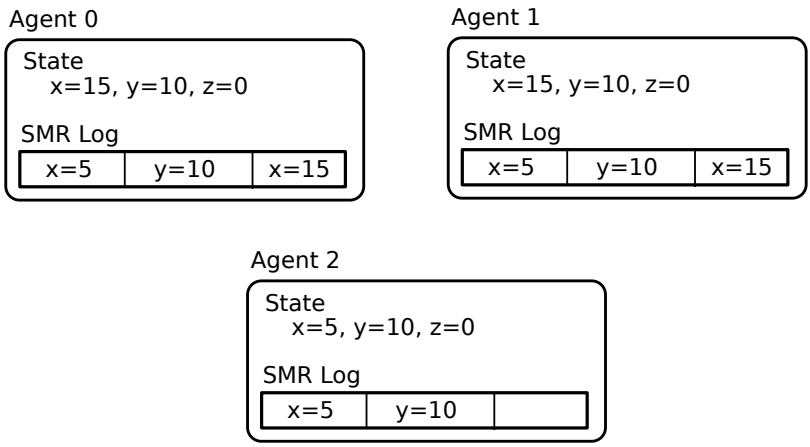


Figure 2.7: An example of a replicated system with three nodes running a SMR protocol. The initial state of each node consists of three variables, x, y, and z, that are initialized to 0. A consensus process is used to write to each log position to ensure agreement and fault-tolerance. In the example, agents 0 and 1 have three entries in their log and the state of the agent reflects processing the first three requests. Agent 2, however, did not receive the third entry yet and its state reflects processing the first and second entries only.

the other agents to decide the c-value for L_i. The decided c-value for L_i is considered the command to be processed when a State Machine reaches position L_i. This c-value may correspond to what the agent received and wanted to commit, or it might be the c-value of another agent. If a command did not *win* a certain SMR log position, the agent can retry with a larger SMR log position.

In summary, by applying a consensus solution to achieve agreement on the content of each position in an SMR log, we are able to ensure that all agents agree on the content of the whole log.

An alternative approach that some protocols follow is to reformulate the problem of consensus for SMR as a problem of building a replicated log [128]. This approach has many similarities to the atomic broadcast problem [71].

3

Background

3.1 Overview and Model of Data Management

Data management systems are complex software that consists of many components and layers [48], [129], [136]. In this monograph, we provide a simple model of data management systems that will help us understand the role of consensus in these systems. This model strips away many components and layers that are not particularly relevant in the interaction between data management systems and consensus protocols. These other components—such as query processing, caching hierarchy, and indexing—are orthogonal to most work that uses consensus for data management.

Our model of a data management system consists of the following two components:

1. *Data storage:* this component represents persistent storage that is used by the data management system. It provides two data abstractions. The first is a key-value store abstraction. The interface of this abstraction consists of a put(in: key, value) and get(in:key; out: value) operations. The second data abstraction is a log. The interface of the log consists of a append(in: entry; out: position)

and read(in: position). We omit the implementation details of the abstractions, including the use of caching mechanisms and garbage collection. A call to put or append immediately leads to persisting the request.

2. *Data access manager:* this component is responsible for scheduling requests and coordination with other agents to ensure the correctness of operation. For example, a data access manager may delay or block certain requests to ensure the absence of concurrency anomalies. The interface of the data access manager is a transactional interface, where a request is a bundle of read and write operations to be processed as a whole (we talk more about transaction processing in the next section.) We assume that the data access manager is the only component with access to the data storage component.

3.2 Transaction Processing, Concurrency Control, and Recovery

A database transaction is a widely-used access abstraction in data management systems. A transaction consists of a collection of read and write operations. A data management system ensures the ACID properties of transactions [57], [72]:

A. *Atomicity:* this is an all-or-nothing guarantee. A transaction either appears fully (with all its operations) to other transactions or not at all.

C. *Consistency:* if the state of the data management system is consistent (i.e., satisfies the correctness invariants of the application), then applying a transaction to it would not lead to an inconsistent state that violate the application's invariants.

I . *Isolation:* transactions running concurrently should not interfere with each other.

D. *Durability:* if a transaction is committed, its effects will always be part of the data management system state.

We talk in the rest of the section on the role of consensus to achieving isolation and durability in distributed data management systems.

3.2.1 Consensus for Isolation

Isolation is a property that manages concurrent execution of transactions. Concurrent execution is possible in different settings. For example, it can occur if the operations of transactions are interleaved within a single machine. It can also occur if transactions are processed across different machines (agents). This is the case that is most relevant to this monograph. When transactions are processed concurrently across agents, there is the possibility of conflicting activity between two agents—e.g., two agents reading and writing the same data objects at the same time. This invites the use of consensus protocols as their purpose is to make distributed agents achieve agreement. As we will see in the rest of the monograph, consensus can be used in different ways for distributed agents to reach consensus on the state of data or operations to ensure the isolation of transactions.

Here, we will present some background on isolation and concurrency control for database transactions. This will aid in understanding how consensus helps in achieving these properties in later sections. We will focus on *serializability* [21], which is a widely studied and adopted isolation guarantee.

At a high-level, serializability aims to create the illusion that all transactions running on the system are isolated from all other transactions. Therefore, from the perspective of a transaction t, any other transaction either appears to has been processed and committed completely prior to the start of t, or is processed and committed completely after the commitment of t. An outcome of this property is that the state of the database reflects a serial order of the execution of the transactions that were applied to it. For example, for a database where two transactions (t_1 and t_2) were committed, a serializable final state of the database must reflect the ordered execution of any of the transactions' permutations: $t_1 \rightarrow t_2$ or $t_2 \rightarrow t_1$. Consider a database with initial state $x = 0$, t_1 is a transaction that updates the value of x to $(x + 1)^2$, and t_2 is a transaction that updates the value of x to $(x + 2)^2$.

A serializable execution of the two transactions would either yield 9 (which corresponds to running t_1 first) or 25 (which corresponds to running t_2 first.) Any other outcome of running these two transactions is not serializable.

Note that serializability is a guarantee of the *appearance* of a serial execution, not that transactions must be processed serially. Therefore, the operations of transactions may still interleave as long as the outcome of processing all transactions appear serial after they commit. This is the attraction of serializability, which is that the guarantee is strong—where a programmer can write transactions as if they are running in isolation—and the possibility of interleaving allows transaction processing to interleave operations to achieve more performance through concurrency.

As an example of allowed interleaving, consider the two transactions t_1 and t_2 that we defined above. Assume that in addition to the operation of reading and updating x that t_1 also writes the value 5 to y and t_2 also writes the value 10 to z. The following is a representation of the operations of the two transactions: $t_1 = \{r_1(x)w_1(x)w_1(y)\}$ and $t_2 = \{r_2(x)w_2(x)w_2(z)\}$, where $r_i(x)$ is an operation of t_1 reading x and $w_i(x)$ is an operation of t_1 writing to x (we omit the value-to-be-written for brevity). Now, consider the following *execution history* that interleaves the two transactions: $h = r_1(x)w_1(x)r_2(x)w_2(x)w_2(z)w_1(y)$. In this execution history, t_1 starts first by reading and writing to x. Then, t_2 performs all its operations before the final operation of t_1. If we start from a state of a database with all data objects set to 0, then the execution history h yields a database with state $\{x = 9; y = 5; z = 10\}$. This state is equivalent to the outcome of serially running t_1 followed by t_2. Because the outcome of h is equivalent to a serial history, h is said to be serializable. On the other hand, a history such as $h' = r_1(x)r_2(x)w_2(x)w_2(z)w_1(x)w_1(y)$ is not serializable.

A method to understand whether operations from different transaction can be interleaved (or reordered) is to observe if they *conflict* with each other. The notion of a conflict describes whether two operations access the same data object and one of the two operations is a write. When two operations conflict with each other, interleaving them has consequences. Because at least one of the two operations is a write, when two conflicting operations are interleaved, the write might influence

the view of the other conflicting operations, thus breaking the isolation guarantee of each transaction. Two read operations do not conflict even if they access the same data object. This is because a read does not change the view of the state seen by the other operation, and therefore, the isolation illusion is maintained.

Using the notion of conflicts, it is possible to have a systematic way to describe a serializable execution of operations. This is done by using a *serializability graph* $SG = (V, E)$, where the vertices V are the transactions and the edges E are conflicts between transactions. A transaction t_i has a directed conflict to t_j in one of the following cases:

- *Write-read conflicts* ($t_i \rightarrow_{wr} t_j$)*:* this conflict exists if there is a write operation in t_i that writes a value that is read by a read operation in t_j.
- *Write-Write conflicts* ($t_i \rightarrow_{ww} t_j$)*:* this conflict exists if there is a write operation in t_i that writes a value that is overwritten by a write operation in t_j.
- *Read-write conflicts* ($t_i \rightarrow_{rw} t_j$)*:* this conflict exists if there is a read operation in t_i that reads a value written by a write operation in some transaction t_k that itself is overwritten by a write operation in t_j.

A conflict between two transactions represents an ordering requirement; if a conflict from t_i to t_j exists, then t_i must be ordered before t_j in any equivalent serial execution.

Using the serializability graph, a serializability check is performed by observing whether a cycle exists. If a cycle exists, this means that the execution is *not* serializable. This is because a cycle denotes that there is at least one sequence of conflicts that starts from a transaction t and ends at the same transaction. However, because a conflict is an ordering requirement, a cycle means that transaction t must be before itself in an equivalent serial execution, which is not possible.

The absence of cycles in the serializability graph, on the other hand, is sufficient to prove that the execution is serializable. Because there are no cycles, the graph can be used to come up with a serial execution

equivalent by traversing the graph and ordering transactions according to the conflict relations.

There are many isolation guarantees other than serializability that range in their level of strictness/flexibility [6]. For example, Snapshot Isolation [6], [20] does not guarantee the equivalence to a serial order. Instead, it guarantees that each transaction operates on a consistent snapshot of the database and that a transaction commits only if there are no write-write conflicts with concurrent transactions.

3.2.2 Replication for Durability

Durability is a property of transactions that the effect of a committed transaction should be always observed even after a crash or unavailability of the whole or parts of the system. In centralized data management systems, this is typically ensured by persisting the state of committed transactions in persistent storage so that they can be recovered after a crash. Distributed data management systems widens the scope of how durability can be achieved. In particular, the state of committed transactions might be recovered from a remote node. Furthermore, highly-available systems take another approach to durability by maintaining different copies of the same data so that even during an agent's crash/unavailability, other agents still have the state of committed transactions.

This later approach is relevant to this monograph and to the use of consensus in data management systems [22], [93]. When multiple copies of the same data are maintained across distributed agents, the agents need a way to coordinate access so that the state of a committed transaction is not *lost* after a crash/unavailability of some agents.

4

Consensus for Distributed Commit

4.1 Overview of Distributed Databases and Atomic Commitment

Distributed data management systems face unique challenges where consensus can be helpful. Distributing a data management system can be done for various reasons. Distributing the state across different agents means that the failure of one agent does not lead to the failure of the whole system. Also, distributing the state may lead to better performance characteristics by distributing the load across different agents.

When the state of the database is distributed across agents, there is a need to ensure that transactions are processed correctly (that they are ACID-compliant.) Achieving ACID guarantees is complicated in distributed data management systems due to the potential for message delays and drops when agents are communicating with each other. For this reason, distributed data management protocols have been developed to ensure ACID properties for transactions across distributed agents.

In this section, we focus on the distributed *atomic commitment* problem. This problem considers a setup of distributed data management systems where data is partitioned and each partition (also called shard) is placed on an agent. Partitions are mutually-exclusive, so each data

object belongs to exactly one partition. An atomic commit protocol ensures that a transaction is processed atomically—the outcome of a transaction is either observed on all corresponding partitions or at none of them.

An atomic commit protocol is typically divided into a commit protocol, a termination protocol, and a recovery protocol. The commit protocol is the protocol that controls how live agents coordinate with each other to commit a transaction. The termination protocol controls how a live agent reacts when it detects the failure of an agent to attempt to complete the commitment of the transaction. The recovery protocol controls how an agent recovers its state after a failure or crash.

We begin by describing popular distributed atomic commit protocols and then show how consensus has been used to solve the distributed atomic commitment problem.

Two-Phase Commit

Two-Phase Commit (2PC) [56], [108] is a distributed atomic commit protocol that ensures that all agents of a distributed data management system agree to commit a transaction before the effects of the transaction is applied to any agent. Typically, an agent—which holds a partition of the data—agrees on committing a transaction if it does not conflict with any transaction it has committed or agreed-to-commit and if it does not lead to a deadlock with other transactions. Agreeing on committing a transaction is also called *preparing* a transaction. Also, an agent might delay preparing a transaction if it is waiting for other events to occur first—for example, if the transaction reads a value that has not committed yet.

The 2PC commit protocol distinguishes between two roles (Figure 4.1): (1) a coordinator that drives the commitment of the transaction, and (2) a participant that represents an agent with a shard that is accessed by the transaction. At a high-level, the 2PC protocol proceeds in two steps. In the first step, the coordinator pull participants to know whether they agree on committing (i.e., preparing) the transaction. A participant is an agent hosting a shard that is accessed by the transaction. The second step happens after the coordinator hears

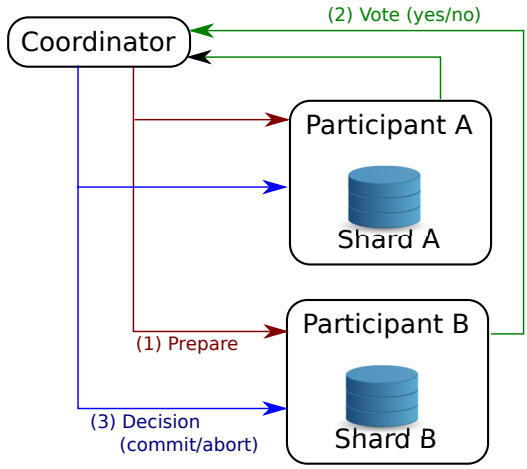


Figure 4.1: A 2PC system with a coordinator and two participants. Each participant hosts a shard (data partition) of the data. The normal-case flow of operations in 2PC begins with a prepare message from the coordinator to the participants. Then, participants respond with yes or no votes. Finally, the coordinator responds with the decision—commit if all participants voted yes, and abort otherwise.

from *all* participants. If all of the participants agree on committing the transaction, then the coordinator sends a commit message to all of them to commit the transaction. Otherwise, if at least a single participant objected to committing the transaction, then the coordinator sends an abort message to all participants.

Note how aborting a transaction is a unilateral decision that can be taken by the participant. This is because committing a transaction at that participant might lead to an inconsistency or deadlock. Therefore, a commit decision cannot be forced on the participant. Another important property of the 2PC protocol is that when a participant prepares a transaction—by responding positively in the first round—this is a promise to be able to commit/apply the transaction if the coordinator sends a commit message. To this end, the participant ensures that future transactions that conflict with a prepared transaction are not prepared or committed.

The steps of a 2PC protocol are shown in the states diagrams in Figure 4.2. The coordinator (at agent 1) starts from the initial state q. When a transaction request is received from the client, the coordinator

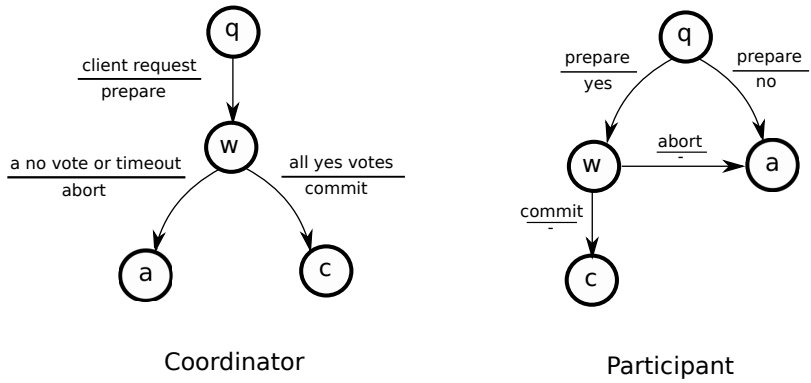


Figure 4.2: A state transition diagram of 2PC. A state with the label q is an initial state, label w is a waiting state, label a is an abort state, and label c is a commit state. Each state transition is denoted by the trigger of the transition (the part over the line) and actions that are performed with the state transition (the part under the line).

broadcasts a transaction commit request to participants, which are agents that maintain shards that are accessed by the transaction. Then, the coordinator waits to hear yes or no responses from the participants. If at least one response was a no, then the coordinator aborts the transaction. If all responses were yes, then the coordinator has the option of either committing the transaction or aborting the transaction. The choice to commit the transaction is the natural choice to make useful progress. The choice of aborting a transaction with all yes votes is to allow the coordinator to unilaterally abort a transaction in the case of a timeout, a client choosing to abort the transaction, or a similar event.

The participant algorithm (as shown for the case of a participant at agent 2) starts in the initial state q. When a transaction commit request is received from the coordinator, the participant decides whether it can commit the transaction. Typically, this entails trying to acquire locks on all accessed data objects. If this is successful, then the transaction is *prepared* and a yes vote is sent back to the coordinator. Because locks are held, this means that no conflicting transaction can commit with a prepared transaction. Otherwise, if locks could not be acquired, then the transaction is unilaterally aborted and a no vote is sent to the

coordinator. If a yes vote is sent back, then the participant waits for the final decision by the coordinator. This is either a commit or abort decision. In both cases, when the decision is received, corresponding locks are released. What is different is that if the decision is commit, then the participant applies the write operations to storage prior to releasing the locks.

Blocking Scenario. One of the challenges faced in the 2PC protocol is that it may lead to blocking scenarios. The reason for this blocking scenario is that a participant with a prepared transaction would not release the resources until it hears the final decision from the coordinator. If the coordinator crashes while the participant is in the prepared state, then the participant blocks until the coordinator recovers from the crash.

At such a scenario, the participant cannot simply abort the transaction while the transaction is prepared. The reason for this is that the participant cannot know whether the coordinator has committed the transaction, responded to the client that the transaction committed, and then crashed before responding to the participants. At this case, aborting a transaction by a participant might lead to an inconsistency between the state of the database and the response to the client. Likewise, the participant cannot commit the transaction if it did not hear from the coordinator, since the coordinator might have aborted the transaction and responded accordingly to the client before crashing.

This blocking problem of 2PC ignited many follow-up works in the database community to propose non-blocking atomic commitment protocols [58], [70], [107], [120], [141], [142]. This includes variants that are based on utilizing consensus. Using consensus—as we will show—turns out to be a good candidate to implement non-blocking protocols. We will motivate the use of consensus and overview solutions adopting this strategy in this section.

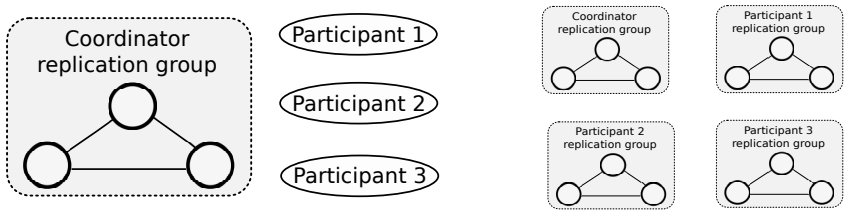


Figure 4.3: An illustration of the two approaches to utilize consensus for atomic commit protocols. The illustration of both approaches contains a coordinator and three participants.

4.2 Consensus for Distributed Atomic Commit

4.2.1 The case for consensus for atomic commit protocols

As we have observed, distributed atomic commit protocols face challenges in terms of blocking scenarios due to agent failures and network partitioning (when network link failures separate nodes into *partitions* and a node in one network partition cannot communicate with any node in another partition.) Some atomic commit protocols overcome these scenarios but result in increasing the overhead and complexity of the protocol. It was then observed that the core of these blocking scenarios and complexity to solve them is that a commit decision might be *lost* after an agent failure or network partitioning. To overcome this problem and to enable *remembering* a commit decision, consensus can be used. Consensus ensures that once a *value* is chosen across participating agents, this value can be recovered by any majority of agents. This insight has been used in various forms to solve the blocking scenarios of distributed atomic commit protocols by using a consensus component. The consensus component is integrated with the atomic commit protocol to solve this problem while maintaining the coordination patterns of the original atomic commit protocol without incurring significant additional complexity.

There are two main approaches to utilizing consensus for the atomic commit protocol (Figure 4.3). The first is to maintain a set of agents to collectively act as the coordinator of the atomic commit protocol. When

a commit decision is made, it is committed to a consensus protocol across these agents. Therefore, as long as a majority of these agents are reachable and running, the decision can be restored. We will call this approach the *resilient coordinator approach*. The second approach focuses on participants of the atomic commit protocol. Each participant of the atomic commit protocol is implemented as a set of agents that collectively act as the participant. When the participant makes a step in the atomic commit protocol, it is first committed to the consensus protocol. Therefore, as long as a majority of agents that represent a participant are reachable and running, the steps and decisions made by a participant can be restored. We call this approach the *resilient participants approach*. Next, we provide more details about the two approaches and present implementations that follow these approaches.

4.2.2 Approach 1: Resilient Coordinator Decision

The resilient coordinator approach ensures that a coordinator's commit decision can always be recovered by replicating it across a set of agents, where a majority of these agents are assumed to be reachable and running. This observation can be traced to early work in making atomic commitment resilient by replicating the state of the coordinator and using a distributed consensus protocol [43], [121].

In its most basic form, this approach adds a consensus step after a coordinator reaches a commit/abort decision but before propagating that decision to clients and other participants. Consider the following sample solution that utilizes paxos as a consensus protocol that is augmented to 2PC. The system model consists of $2f + 1$ coordinators and N participants, where N is the number of partitions. The $2f+1$ coordinators run a paxos protocol. When a request to commit a transaction is received it is written to the paxos log using the paxos algorithm. Once the transaction information is written to the log, it can be safely recovered as long as a majority of coordinators are available.

The leader that wrote the transaction to the paxos log then sends a prepare message to the participants. Each participant follows the 2PC protocol and responds with a yes or no message. When the leader makes a decision of whether to commit or abort, it writes that decision to the

paxos log. Once written, the decision is propagated to the participants and client.

This design ensures that the information about the transaction is preserved despite the failure of a coordinator agent, due to writing the transaction information in the paxos log initially. The design also ensures that the commit/abort decision is preserved despite the failure of a coordinator agent, due to writing the decision in the paxos log before propagating it.

If a coordinator crashes at any time, another coordinator can pick up and recover the commit/abort decision if it has been made. Consider the case of the failure of a leader coordinator. This failure can happen in one of the following two cases:

- Before the transaction information is written to the paxos log: In this case, a decision has not been propagated to participants or clients, since sending the decision to participants and clients need to happen after persisting the decision in the log. In this case, the new leader can unilaterally abort the transaction. It does so by first writing the abort decision to the paxos log. If it is written, then the new leader propagates the decision to the participants and client.

- After the transaction information is written to the paxos log: In this case, the new leader recovers the decision whether a commit or abort decision. Then, it propagates that decision to the participants and client.

In both cases, if a failure happens to the new leader, then another leader is elected that performs the same recovery steps.

4.2.3 Approach 2: Resilient Participants

The second approach is to make the decision made by the participant be resilient. Therefore, when a participant makes a yes or no decision, it is made resilient before it is propagated to other agents.

Paxos Commit. This basic approach has been adopted by various solutions. One of the early examples of this approach is the Paxos Commit algorithm [58]. In Paxos Commit, a paxos cluster is used

to obtain agreement on the decision of each participant (this cluster can be colocated with participants.) The paxos cluster maintains n paxos instances, one for each participant, where n is the number of participants. Each participant writes their decision (yes or no) in the paxos instance that corresponds to them. The transaction commits if all participants wrote yes. Otherwise, the transaction aborts. The transaction coordinator (or any other agent) can infer the outcome of the transaction by observing the values written by the participants.

If one of the participants experienced a failure, then a decision is not written in its paxos instance. The coordinator, after the expiration of a timer, suspects that the participant failed and tries to write a no decision in the participant's paxos instance. In this attempt to write no, the coordinator either learns that a decision has been written, in which case it considers that decision, or it successfully writes no on behalf of the participant. This step is possible because a decision has not been written to the paxos instance, and thus would not have led to a decision that conflicts with the unilateral no decision.

Paxos Commit adopts this basic approach of utilizing a consensus instance for each participant. Any consensus protocol can be used in this approach. However, it is possible to optimize the number of message exchanges by considering a specific consensus protocol. Paxos Commit discusses how messages can be saved when using the paxos protocol as the consensus component of the Paxos Commit protocol.

Spanner. Another example of the resilient participants approach is Google Spanner [40], a distributed multi-data center data management solution. We focus on the distributed transaction commit protocol in Spanner that builds on the 2PC protocol. In Spanner, data is partitioned into n partition. Each partition is maintained by a cluster of geo-replicated nodes, where the number of nodes in the cluster is $2f + 1$. Each one of these nodes is replicated in a different data center to tolerate data center-scale failures.

Within a shard's cluster, paxos is used to replicate the steps that are taken to process 2PC requests. Specifically, the paxos leader in that cluster processes the requests that correspond to the 2PC participant of its shard. By replicating the steps of 2PC, a failure of a shard's leader can be tolerated by electing a new leader that reconstructs the state of the participant and uses it to answer incoming requests.

The protocol proceeds in the following way (Figure 4.4). When a new transaction is ready to commit, a coordinator—which can be co-located with a shard leader—first logs the prepare message information in stable storage. Then, it sends a prepare message to the paxos leaders of the accessed shards. For example, if there are three accessed shards, then the coordinator sends a prepare message to three nodes, where each node is a leader of one of the accessed shards.

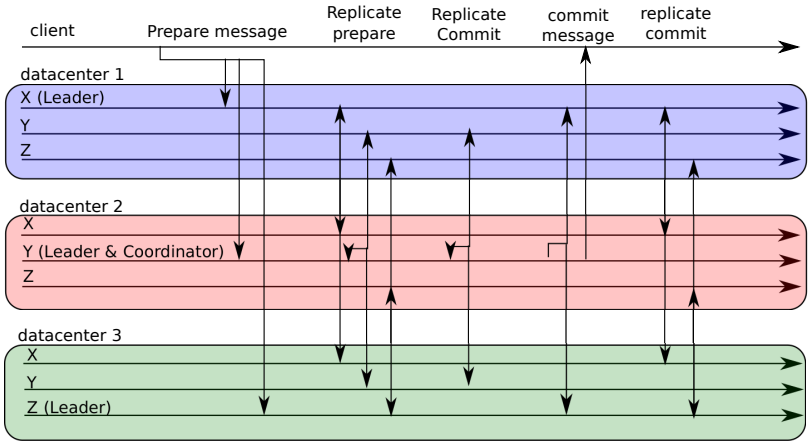


Figure 4.4: A schematic diagram of the operation of Spanner in a scenario with three data centers each hosting three shards, X, Y, and Z.

When a shard leader receives a prepare message, it acquires the corresponding locks for the transaction to be committed and it logs the prepare message information in persistent storage. Afterwards, the shard leader replicates the prepare information to the other nodes in the cluster using the shard's paxos instance. Then, the leader waits for the information to be successfully replicated (i.e., accepted by a majority of nodes). After a leader replicates the prepare information successfully, it responds to the coordinator acknowledging that their shard has prepared the transaction.

The coordinator waits until it receives prepare acknowledgments from all shards' leaders. Once it does, the coordinator commits the transaction locally by writing the commitment information to persistent storage. Then, the coordinator replicates the information using paxos

to a majority of nodes. Once the commit information is replicated successfully, the coordinator sends commit messages to the leaders of accessed shards. At this point, the client can be notified that the transaction has committed.

When a shard's leader receives the commit message, it persists and replicates the information to a majority of nodes in the shard's cluster. Once the replication round is successful, the shard's leader releases the locks.

Replicated Commit. The next example we present for this approach is of Replicated Commit [114]. Replicated Commit builds on the Spanner protocol and aims to reduce the number of cross-data center messages by rearranging the patterns of replication. Specifically, instead of each 2PC participant replicating the 2PC steps, the whole transaction is replicated and processed locally at a data center and then global coordination is performed to ensure that the outcome is both safe and fault-tolerant.

The system model of Replicated Commit is similar to Spanner. Data is partitioned into n partitions. Each partition is maintained by a cluster of $2f + 1$ geo-replicated nodes to tolerate data center outages. We assume that data is fully replicated in each data center for ease of exposition.

The Replicated Commit protocol proceeds in the following way (Figure 4.5). An application client—client for short—drives the commitment of the transaction. When the transaction is ready to commit, the client replicates the transaction commit request to one of the accessed shards. The shard can be chosen arbitrarily and will act as the coordinator of the commitment (in each data center, the node of the chosen shard acts as the coordinator of that transaction commit request).

When a coordinator node receives the commit request after it is replicated, it forwards the commit request—as a 2PC prepare request—to the other cohorts within the same data center. The other cohorts are nodes that correspond to shards accessed by the transaction in the data center. Note that this step is performed locally at the data center for all data centers, which means that no wide-area messages are exchanged.

When a cohort node receives the prepare request, it acquires the corresponding locks and logs the prepare information in persistent storage.

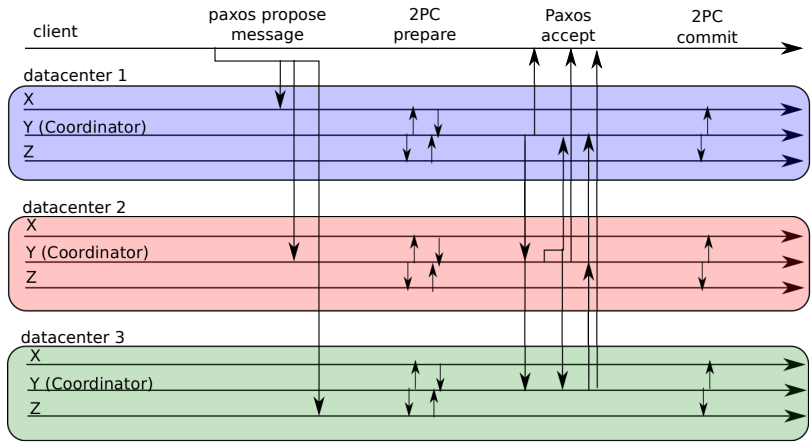


Figure 4.5: A schematic diagram of the operation of Replicated Commit in a scenario with three data centers each hosting three shards, X, Y, and Z.

Then, they respond to the coordinator in their data center—which is also an intra-data center message exchange.

When a coordinator in a data center receives acknowledgments from all its cohorts, then it accepts the commit request by sending an accept message to other coordinators as well as the client. When a coordinator receives a majority of accept messages, then it considers the transaction has committed and sends the commit decision to the cohorts in its data center. When a node receives a message that the transaction has committed, it releases all corresponding locks.

4.3 The Relation Between Atomic Commitment and Consensus

The effectiveness of protocols that utilize consensus to solve the distributed atomic commit problem inspired work to explore the similarities of goals and operation of the two problems [59], [115], [123], [162]. Guerraoui [59] shows that the problem of non-blocking atomic commitment is harder than consensus. The Consensus and Commitment (C&C) framework [115] finds a model that unifies the operation and steps of consensus protocols (such as paxos) and commitment protocols (such as 2PC). The observation that is made by that work is that there are steps in both protocols that aim to achieve the same goal. Specifically,

the unification model consists of four phases (for ease of exposition we project the model phases to protocols based on paxos and 2PC rather than general consensus and atomic commit protocols):

- *Leader election:* The coordination in both paxos and 2PC are driven by a single node; a leader in paxos and a coordinator in 2PC. This phase captures the similarity in the selection of a leader in paxos and the selection of a coordinator in 2PC.
- *Value Discovery:* Both paxos and 2PC aim to reach agreement; paxos on an arbitrary value, and 2PC on whether to commit or abort a transaction. This phase aims to capture the similarity of how both protocols need to discover what are the candidate values to be committed. In paxos, this is done in the leader election phase where agents respond with previously proposed c-values. In 2PC, this is done by collecting yes and no votes.
- *Fault-tolerant agreement:* Both paxos and 2PC protocols emphasize fault-tolerant agreement. In paxos this is done by replicating proposals and ensuring that a proposal is not committed until it is replicated. In many 2PC variants and follow up work, the votes and/or decisions are replicated for fault tolerance.
- *Decision:* This phase captures the propagation of the decision from the leader/coordinator to other nodes after commitment.

Understanding the similarities of consensus and atomic commit protocols can help in understanding these protocols better. Also, they can enable finding better protocols that combine aspects of both protocols [115], [123], [162].

5

Consensus for Data Replication

5.1 Overview of Data Replication

An area of data management where consensus has a significant impact and role is the area of data replication. Data replication describes the feature of replicating the state of data across multiple distributed nodes, where each node has a copy of the data. The copy at a node can be a full copy or a partial copy (i.e., containing a subset of the data.) For ease of exposition, we assume full replication throughout this section.

5.1.1 Benefits of Data Replication

Data replication is used to increase the resilience of data in the face of machine faults. When there is more than one copy of the data, a failure of one node—hosting one of the copies—would not prevent the progress of the application if another node—hosting another copy of the data—takes over and continues the operation. Consensus mechanisms are helpful for this function as they enable making progress with a subset of the nodes, which enables tolerating the failure of a subset of nodes. For example, as we have discussed when presenting paxos, progress can be made with only a majority of agents. Therefore, if any

group of nodes fail, progress can still be made as long as the number of failures is less than a majority.

Another benefit of replication is increasing read availability, where data is accessible from more than one node. This enables clients to send requests to different nodes, reducing the load on each replica. Load balancing techniques can be used to ensure that the participating nodes receive proportional workload. Consensus is critical for this function as it helps coordinate the different copies so that they are consistent with each other. Without a method to coordinate access, it is possible that different nodes may receive conflicting and/or inconsistent outcomes for their read operations.

Replication may also be performed to scale the performance of the system. This is not always the case with replicated systems, since increasing the number of nodes leads to increasing coordination overhead which degrades performance. However, in certain cases, and when done carefully, replicating the state of an application may help scale performance. This is the case for operations that do not require extensive coordination such as read and commutative operations. Another case where scalability is possible through replication is when nodes work collaboratively to process requests, which reduces the workload on each node.

5.1.2 Replicating Databases

Database replication has been an active area of research for many decades due to the benefits of replication. There are various ways to categorize replication strategies for data management systems. Next, we provide a quick overview of some of them and refer the interested reader to more comprehensive surveys [89].

Primary-copy vs. update-anywhere replication. Primary-copy replication approaches designate a single agent as a primary and all other agents as secondaries. The purpose of this designation is to make the primary the point of entry to the replication process. An operation is first sent to the primary to be processed and then replicated to other secondary agents. The advantage of this approach is that it simplifies the coordination protocol. However, the downside is that the primary

becomes a single-point of failure. Primary-copy approaches overcome this challenge by providing mechanisms to detect a primary failure and then designate one of the secondary agents as the new primary. In terms of the performance bottleneck, existing systems enable serving requests through secondary nodes in special cases—by relaxing consistency [16] or utilizing special types of operations [118].

On the other hand, the update-anywhere replication approach is different in that it allows operations to be sent to any replica that manages the operation's execution. This approach overcomes the single-point of failure problem of primary-copy approaches. However, it adds to the complexity of the design as agents need to coordinate with each other to ensure that their copies are consistent. Without a single-point of synchronization, this coordination can be become expensive, requiring extensive coordination and/or gossip messages.

Eager vs. lazy replication. Eager replication approaches return a response to the client only after the client's request is successfully committed and replicated to enough replicas to tolerate f failures. The eager approach ensures that the client gets the response after the operation is performed and committed, avoiding consistency anomalies. However, this means that a client must wait for the necessary coordination to be performed first which leads to the potential of high latency. Lazy replication, on the other hand, returns the response to the client immediately and performs the replication process *lazily* in the background after the response is sent back to the client. This enables returning a response to the client fast. However, in the presence of an inopportune conflict or failure, it is possible that the response to the client is erroneous or inconsistent with other client responses.

In some literature, eager replication is termed synchronous replication while lazy replication is termed asynchronous replication. These terms, however, are not connected to the communication model meanings of synchronous and asynchronous communication.

Read and write quorums. Some replication protocols for data management systems take the approach of defining read and write quorums. A read quorum is what a read operation uses to serve a read operation, while a write quorum is what a write operation uses to serve a write operation. The insight behind this approach is that if the two

quorums intersect, then this means that a read quorum will always be able to see a previously committed write—if the two quorums intersect, then there is at least one agent that is in both the read and write quorums. A common approach for defining read and write quorums is to make them majority quorums which ensures intersection. Other approaches differentiate between read and write quorums based on the workload and the priorities of the application. For example, in a read-heavy workload, it is beneficial to make the read quorum small even at the expense of a larger write quorum to ensure intersection.

One of the issues that are faced in this approach is that multiple write operations may be delivered differently on different agents. For example, in a system with three agents, a_0, a_1, and a_2, consider that a_0 issues a write w_0 that updates the value of x to 0, while a_1 issues a write w_1 that updates the value of x to 1. If the two writes are concurrent, it is possible that a_0 orders its write before w_1 and a_1 orders its write before w_0. In such a case, readers may have a conflicting view of the value of x based on which agents they asked. To overcome this problem, a write rule needs to be applied to ensure that the values converge to a single value in the absence of new writes. A common write rule is *Thomas' Write Rule* [146] in which each write operation has a unique sequence number—for example, a combination of a timestamp and the source node's unique id. Each node maintains the write operation with the higher sequence number regardless of the order in which write operations are received.

5.2 Consensus for Data Replication

5.2.1 The case for consensus for data replication

The overview in the previous section shows how data replication protocols aim to maintain copies of data across distributed agents with two main goals: (1) ensure that the copies are consistent, and (2) ensure that agent failures are tolerated (up to f failures). These two goals are similar to the goals of distributed consensus protocols such as paxos. A consensus protocol aims to ensure agreement across distributed agents which corresponds to the first goal. Additionally, consensus protocol reach agreement while tolerating f agent failures. This observation has

led many system designers to explore the use of consensus protocols as a basis to solve the data replication problem.

5.2.2 State-Machine Replication

The most common way to utilize consensus for data replication utilizes the state-machine replication (SMR) approach (see Section 2.3 for more information about SMR and how consensus is used to solve the SMR problem.) In this case, a data management system writes data operations or database transactions in the SMR log. The SMR log acts as the ordering mechanism of database transactions, where the order of transactions in the SMR log would represent the execution order in all replicas. By following the same order of transaction execution, the state of replicas is guaranteed to be consistent. The following are examples of systems that follow the approach of utilizing consensus and state-machine replication for data replication.

Megastore

Megastore is a data management system proposed by Google [17]. We focus on how they manage the replication of a partition of data which is relevant to this section. Megastore's goals are to provide transparent recovery from failures without sacrificing strong consistency (i.e., ACID transactions.) For this reason, they avoid the use of lazy (asynchronous) replication strategies as they may lead to inconsistencies. Also, they avoid the use of protocols that require a "heavyweight" primary node as "failover requires a series of high-latency stages often causing a user-visible outage." Alternatively, Megastore utilizes paxos to manage replicating state as it allows maintaining strong consistency as well as transparent recovery from failures. Transparent failure recovery is achieved because paxos does not distinguish a *failed* state which would require additional complexity in terms of detecting it and reacting to it.

In Megastore, a partition is replicated across a number of agents. The agents maintain a SMR log where paxos is used to commit entries to the SMR log. An entry represents a commit record of a transaction that includes the write/update operations of the transaction. Therefore, the SMR log acts as a write-ahead log of transactions. Megastore

applies a form of multi-version concurrency control (MVCC) to commit transactions. Each transaction is timestamped and its corresponding write/update operations have the same timestamp of the transaction that wrote them.

A transaction client (client for short), reads the state of the data as of the timestamp of transaction that was most recently written in the SMR log. It continues processing the transaction locally by reading as of the assigned timestamp and buffering writes. When the client is ready to commit a transaction (all reads and buffered writes are complete), it starts the transaction commit process. This process starts by assigning the transaction a commit position in the SMR log. This position is the position that is right after the position that the transaction read from. For example, if the transaction read as of the timestamp of a committed transaction in log position i, then the assigned commit position is log position $j = i + 1$.

The client acts as a paxos proposer and attempts to write the commit record of its transaction to SMR log position j. This attempt might succeed—in which case the transaction is considered committed—or fail—in which case the transaction is considered aborted and the client may retry/restart the transaction.

Advisory locking. One of the challenges faced in the commit process of Megastore is that all concurrent transactions compete on writing to the same log position. For example, consider two concurrent transactions t_1 and t_2 that both are reading as of the same log position i. Since both transactions would compete to writing to the same log position j, only one of them can commit. If there are n concurrent transactions, then only one can commit and the rest $n - 1$ transactions abort. This leads to wasted overhead since transactions would restart processing every time they are aborted. To avoid this wasted overhead, advisory locks can be used to sequence transactions so that they are processed back-to-back.

Although advisory locks help in reducing wasted overhead, there is still a limit on throughput since only one transaction is processed at a time. To avoid this, Megastore apply bulk processing techniques that allow batching transactions together. This allows writing a batch of commit records collectively which increases the aggregate throughput.

Local reads. Megastore allows reads to be local, thus not incurring the overhead of writing to the SMR log. This is possible because Megastore replicates writes to all replicas. There are three types of read operations: (1) current read: this read is the most consistent out of the three. it first ensures that the writes of all committed transactions are applied to the state of the database and then it reads from the database. (2) snapshot read: this read operation reads as of the most recent applied transaction. Therefore, it is possible that it misses the writes of a committed transaction that is not applied to the database yet. (3) inconsistent read: this read operation does not consider the state of the log or committed transactions, but rather, it reads the latest state of the data directly. Inconsistent reads are useful when there are stringent latency requirements and no strong consistency requirements.

Single round-trip writes. A write to the SMR log may incur two round-trip latencies to a majority of replicas—one for leader election and another for replication. It is possible to avoid the extra round trip and write to the SMR log in one round-trip latency. This is possible by applying methods similar to multi-paxos where an agent performs the leader election phase for many log positions at once. In Megastore, a similar approach is performed. When an agent writes to a log position, it is considered to be the leader of the next log position. This can be thought of as piggybacking the leader election round of log position $i + 1$ with the replication round of log position i.

Paxos-CP

Paxos-CP [131] is a replication protocol that builds on Megastore to increase its concurrency. Megastore, as we described above, faces a limit of committing transactions back-to-back which limits concurrency and throughput. Paxos-CP proposes two techniques to increase concurrency: Combination and Promotion.

Combination. The combination technique explores opportunities to combine concurrent transactions into one SMR log position. Specifically, when a client (i.e., proposer) is attempting to write a commit record for its transaction, it first observes the previously accepted values that are reported in the leader election phase of paxos. By observing the

responses, the client can deduce whether it is possible that any one of these values could have been decided in a prior round. If a value could have been decided, then the client must proceed with that value to avoid conflicts. If not, then the client is free to choose any value in its replication phase. Because of this, the client can combine the transactions that were reported in the leader election phase (ones that are concurrent to the client's transaction), and send a propose message to decide the combined set of transactions together. The client constructs the combined entry so that there are no conflicts between them, i.e., if there is a conflict between two transactions, then one of them is discarded.

One of the critical steps of the combination algorithm is to decide—after collecting prepare-ack messages—whether the client can freely choose a c-value for the replication phase. To do so, the client tests whether each received c-value could have been committed in a prior round. For example, in a scenario with 5 agents, consider the case when a client receives 4 prepare-ack where two agents report accepting the c-value x. In this case, x could have been committed in a previous round because it is possible that the fifth node—that did not respond—has also accepted x, which means that a total of three agents (i.e., a majority) could have accepted the c-value. Consider another case where no more than one agent has reported accepting the same c-value out of the four responses. In this case, it is impossible that one of these four c-values has been chosen, since regardless of the state of the fifth node, no c-value could have acquired a majority of acceptances. In this case, the client may combine commit records for the replication phase.

For combination to work, there is a need to wait for more agents than a simple majority. This may add latency to execution. However, in some cases—when the round-trip latency between agents is similar—responses from more agents are received close to each other. This means that the latency overhead is not always significant. Another overhead that is important to note is verifying what transactions can be combined with each other, and finding the combination set of transactions that would maximize concurrency. This can be a challenge when the number of transactions is large or if batching is performed by each client. In such cases, best-effort approaches might be more suitable that deriving the optimal set of combined transactions.

Promotion. The promotion technique aims to overcome the wasted overhead incurred when a transaction fails to write to its assigned log position. For example, a transaction that reads as of log position i cannot commit if another transaction has already been written to log position $i+1$. In Megastore, such a case leads to the need to restart the transaction which would require processing the transaction again, leading to wasted overhead. The promotion technique allows a client to explore whether it is possible to *reuse* the processing that is already done instead of redoing it. Specifically, the client observes the transaction that was written in the log position it wanted to write to, e.g., log position $i+1$ in the example. If there are no conflicts between the transaction and what is written in $i+1$, then the client *promotes* the transaction to the next log position $(i+2)$ and attempts to write the commit record to it. In this case, there is no need to reprocess the transaction since the transaction in log position $i+1$ does not conflict with it. This is because the transaction in $i+1$ did not change any of the data objects that were read by the client's transactions, whcih means that the view of the database as of log position $i+1$ is identical to the view as of log position i from the perspective of the transaction that is now committed in log position $i+2$.

Atomic Broadcast for SMR

The use of atomic broadcast solutions—which are closely related to consensus solutions—have also been used to implement SMR systems [7], [24], [86], [133], [134], [145]. Atomic broadcast offers reliability, atomicity, and ordering properties. These can be used to implement a SMR system by utilizing atomic broadcast to order messages. Then, this message order will be used as the SMR log order.

5.2.3 Consensus to Manage Access to Shared State

One of the common uses of consensus is to coordinate access to shared resources and distributed state. A distributed system needs to maintain configuration information that allows different agents to learn about the distributed systems and the other nodes. This configuration information is a form of a shared/distributed state. Given the critical nature of

such distributed state, it is desirable to maintain this information in a reliable way. Using consensus protocols help in providing a reliable mechanism to maintain shared and distributed state.

A common use case for such services is to elect a leader for a certain task. For example, consider a distributed processing system where each task must not be processed by more than one agent. A consensus protocol to coordinate access can be used to assign the task to one agent ensuring that no other agent will attempt to process that event. In this section, we present an example of consensus-based systems to manage shared state and discuss its use cases.

Google Chubby [31] is a lock service that aims to coordinate access to shared state in a distributed system. It is a Google service that has been used internally in multiple systems. For example, the Google File System (GFS) [52] uses Chubby to elect a leader for the cluster. BigTable [37] uses Chubby to allow a leader to track the servers it controls.

The interface provided by Chubby is similar to a filesystem with the addition of advisory locks. To coordinate shared state, an agent creates a *file* that represents the state it wants to coordinate. All agents wanting to perform an operation related to the shared state would have to coordinate through the designated file. To facilitate this coordination, reading and writing to the file and advisory locks are used where there are write (exclusive) locks and read (shared) locks. If an agent acquired an exclusive lock, no other agent can acquire the same lock. However, shared locks can be held concurrently.

Consider a system that aims to maintain meta-information about a database that is sharded. The meta-information should enable a client to know the scope and location of data shards. A filesystem interface of the lock service may abstract this information by creating a directory /ls/database-shards/shards/ and a file for each shard inside the directory. Shard i is in file /ls/database-shards/shards/i and consists of the shard's information such as data that is maintained in the shard and the location of the agents that store the shard and process its corresponding requests. When a client wishes to access a shard it acquires a shared lock during the operation. To reconfigure shards, an exclusive lock is used to modify the content of relevant files in the duration of the reconfiguration process.

This locking strategy prevents the case of a reconfiguration while a client is sending requests to agents that are being reconfigured.

Reliability and high-availability of the lock service is one of the main features of the Google Chubby lock service. If the meta-information is maintained in an agent that undergoes a failure, then this would influence all the systems that are managed by it. Making the lock service reliable is important for the reliability of all the services managed by it.

To achieve reliability, paxos is used to replicate the state of the lock service to a cluster of agents. Using paxos ensures that a failure is tolerated transparently by electing a new leader. Such a cluster is called a *Chubby cell* that consists of a number of agents, typically five agents. One of the agents in the Chubby cell is elected to be a leader using the leader election mechanisms of paxos. The state maintained is that of a database that is replicated across agents in a Chubby cell. Read and write requests are sent to the leader to be served. A read request is served by the leader without further coordination if the leader has an unexpired read lease. A write request is first replicated from the leader to a majority of agents in the Chubby cell before a response is sent to the client. This replication step is performed by the replication round of paxos.

Other lock services implement the same mechanisms of using paxos-based protocols to achieve reliable state maintenance. This includes Apache ZooKeeper [83], which is an open-source solution that has been used widely in different services. ZooKeeper shared with Chubby various characteristics such as the use of an agreement protocol to guarantee consistency and the use of a filesystem-like format. One distinguishing feature of ZooKeeper is its support for wait-free data objects. This overcomes the challenges of blocking primitives such as locks.

5.2.4 Consensus in Replicating Participants of Distributed Systems

The use of consensus to maintain replicated state is not only used when replicating the whole application. In many cases, consensus protocols are used to replicate a subset of a larger distributed protocol. An example of this is distributed commit protocols. Instead of replicating the state of the whole distributed commit protocol, an alternative approach is

to replicate the state of each participant. In such a case, the failure of an agent is tolerated because each participant's state is maintained by a cluster of agents. We have discussed this approach in more detail in Section 4.2.3.

5.3 Consensus in Geo-Distributed Systems

One of the main areas of research that utilizes consensus for data replication is the area of geo-distributed systems. A geo-distributed system is a system where data is distributed across geographically distant locations. Typically, data is distributed across multiple data centers. Separating the different pieces of data are wide-area links that incur latencies from 10s to 100s of milliseconds. For ease of exposition, we focus on fully-georeplicated systems across data centers, where each data center maintains a full copy of the data.

The main motivation for geo-replication is to tolerate data center-scale outages [60]. To this end, using a consensus protocol for replication ensures that the outage of a data center can be tolerated by resuming operation using the copies at the other data centers. What distinguishes the problem of geo-replication from regular intra-data center replication is the wide-area latency that separates replication agents. This makes coordination expensive in terms of latency. The evolution of geo-replicated systems started with flat geo-replicated systems that mimic the pattern of regular replication systems. Then, to overcome the high latency, various methods were performed to reduce the number of needed communication rounds. Finally, methods that utilize locality and hierarchy are proposed to avoid large wide-area latency. In the rest of this section, we overview these approaches.

5.3.1 Flat Geo-Replication

Flat geo-replication represents the first wave of geo-replicated systems that mimic the communication patterns of regular replication systems. In these systems, agents are treated equally regardless of their location. An example of this approach is Megastore [17] that we introduced in Section 5.2.2. Flat geo-replication has the advantage that existing

replication mechanisms that were utilized within the data center can be extended with relatively less effort to geo-replication. This led to the adoption of this replication strategy by systems that focus on other aspects of data management. Calvin [147], for example, is a deterministic transaction processing layer that extends the use of paxos to geo-replicate the state of the data management system.

To reduce the wide-area latency cost, flat geo-replication systems employ multi-paxos to avoid the leader election phase latency. They also utilize leader and read leases to enable responding to read requests without coordination with other data centers.

Placement for Flat Geo-Replication

Placement is an important aspect of practical deployment of geo-replicated systems that controls a trade-off between performance and the level of fault-tolerance. Placing data copies across nearby data centers improves performance but may still be susceptible to data center outages impacting multiple data centers such as outages due to natural disasters. Placing copies across faraway data centers would lead to higher latency but reduces the possibility of multiple data center outages[1].

Additionally, the placement of copies across data centers influences the latency based on the dynamics of the protocol. For example, a paxos protocol's performance would be influenced by the latency to the closest majority of data centers. A distributed commit protocol's performance would be influenced by the latency to get to the participants of a transaction being committed.

Solutions to aid in placement of data copies across data centers have been proposed [1], [2], [140], [156]. SpanStore [156] is a placement system that decides where to place copies of data of a key-value store to improve performance and monetary costs savings. It utilizes information about the wide-area latency between data centers, and the costs of storage, processing, and network bandwidth. SpanStore uses an optimization

[1]Note that there are types of outages that can impact multiple data centers regardless of where they are placed. These are outages that are due to configuration and software errors that can propagate from one data center to another [60].

formulation that takes all the input information about the system model, system environment, and workload to produce a placement strategy to maximize the performance and monetary costs objectives while staying within the limits set by the application.

Take me to your leader! [140] is a placement system that considers a strongly consistent transactional system similar to the distributed atomic commit protocols we discussed in Section 4. The database is sharded and each shard is replicated independently. The replication of a shard is performed through the leader of that shard. The goal of this work is to optimize the following aspects of the geo-replicated system: (1) Leader placement: this aspect decides the location of the leader to maximize performance of database operations. The optimizer can choose any replica to become a leader. This optimization considers the locations of clients, their workloads, and the performance of serving their requests. (2) Leader and replica roles: this optimization decides what agents to assign as *voter nodes* that participate in the replication protocol (e.g., acceptors in a paxos protocol) and what agents to assign as *read-only nodes* that would only receive updates and make them available for read operations. This is needed in cases when it is desired to maintain additional agents for read-only operations in addition to the main agents used for replication and voting. (3) Replica locations: this optimization decides what agents to choose out of the pool of all available agents to participate in the protocol either as voter or read-only nodes.

GPlacer [159], [160] is a placement and configuration system that targets multi-data center scenarios. Specifically, GPlacer models the problem of placement as deciding the placement of n replicas across all available data centers. To make this decision, GPlacer considers the topology of the data centers and the wide-area latency between them to estimate the average latency of transactions. For each placement scenario, the average latency can be calculated using the anticipated workload characteristics and origin. For example, for a certain placement, the latency of transactions—whether read-write or read-only—can be estimated for each data center. If this estimation of average transaction latency is performed for all possible scenarios, then it is possible to pick the best placement strategy by choosing the scenario that yields the

best average latency. The space of placements can be large, and for this reason, GPlacer utilizes an optimization framework to reach the decision of the best placement faster.

Reduced Communication Complexity for Flat Geo-Replication

An area of work to reduce the performance overhead of flat geo-replication aims to find ways that would reduce the number of communication rounds beyond what multi-paxos can achieve. There are three main communication overheads that influence the performance of geo-replication systems: (1) The overhead to elect a leader. This overhead can be reduced with the use of multi-paxos as we discussed in Section 2.2.1. (2) The overhead to replicate a c-value. This is the overhead that corresponds to the replication phase of paxos. So far, this overhead corresponded to the latency needed to reach a majority of agents from the leader. (3) The communication overhead of client-leader communication. This overhead is not part of the consensus process itself. However, it is still experienced by the client. This latency can be non-trivial in geo-replication systems. For example, if the leader is in Asia and the user is in the Americas, the wide-area latency between the two continents adds to the client's end-to-end latency both for transmitting the request to the leader as well as receiving the final response from the leader. Note that this overhead is incurred even if there is an agent (e.g., paxos acceptor) that is closer to the client. This is because all requests need to go through the leader.

Protocols in this category aim to manage the trade-offs and design choices of the above three sources of overheads. One approach is to utilize a *leaderless* consensus protocol instead of a leader-based protocol. Variants such as Fast Paxos [101] can be used to allow a client to commit a c-value by sending them directly to acceptors. At a high level, Fast Paxos operates by making clients send a special propose message to acceptors. When a quorum of acceptors accepts the proposal, then it is considered committed. There are some caveats for this approach to work. The client needs a *super majority* to form a quorum, instead of a simple majority. Specifically, in Fast Paxos, there are two types of quorums. A fast quorum that is used to commit c-values from clients to

acceptors directly without a leader, and a classical quorum that is used if a conflict occurred when using a fast quorum. The condition on the fast and classical quorum is the following: any two fast quorums and a classical quorums must intersect in one acceptor. With 5 acceptors, a possible configuration sets the size of the fast quorum to be 4 acceptors and the size of the classical quorum to be 3 acceptors. Any two fast quorums and a classical quorum intersect in at least one acceptor. In the best case, the client can commit a c-value with one round to the super majority (using a fast quorum). However, it is possible that if two clients tried to commit c-values concurrently that neither would collect enough votes, in which case the protocol falls back to using a classical quorum which leads to additional overhead.

The advantage of leaderless protocols in wide-area settings can be significant. The latency of collecting additional votes to form a fast quorum can be much faster than the additional latency needed to funnel requests through a potentially distant leader. For this reason, geo-replication solutions explored the use of such protocols. Multi-Data Center Consistency (MDCC) [95] is a protocol that explores the use of Fast Paxos in geo-replication. Clients in MDCC commit database transactions by replicating them to acceptors using Fast Paxos. They observe that a high conflict rate leads to repeated fallback to classical paxos. Not only is the benefit of Fast Paxos not observed in such a case, there is extra overhead for conflict detection and fallback to classical paxos which means that the performance becomes significantly worse than running a typical multi-paxos protocol.

To overcome the challenges stemming from workload with high conflict rates when using fast paxos, MDCC exploits cases when operations are commutative to avoid conflicts [118]. Commutative operations are operations that can be reordered without impacting the final outcome of applying the operations. In another way, commutative operations ensure that they do not conflict even if they touch the same data objects. For example, incrementing an integer is a commutative operations, since two increments can be reordered without invalidating their computation even if they operate on the same variable. MDCC allows committing commutative operations even if they are applied concurrently by utilizing Generalized Paxos [100]. In Generalized Paxos,

c-values are committed in a similar way to Fast Paxos. However, instead of requiring that acceptors agree on the order of committed c-values, it allows an acceptor to accept c-values in any order as long as the operations in the c-values are *compatible*. Compatibility can be defined by the application of Generalized Paxos. MDCC utilizes the notion of compatible operations to support committing commutative operations concurrently. Therefore, MDCC allows committing c-values using fast quorums while avoiding conflicts by using commutative operations.

Leaderless protocols based on paxos have been proposed to optimize the size of the fast quorum further. Egalitarian Paxos (EPaxos) [122] is a leaderless protocol that reduces the size of the fast quorum to be $f + \lfloor \frac{f+1}{2} \rfloor$ out of $2f + 1$ total nodes. The novel design that allows EPaxos to achieve such small quorum sizes is that acceptors respond with dependency information of requests they receive. This allows tracking dependency information by clients which enables resolving conflicts by ordering them when possible. When an order cannot be guaranteed—because multiple nodes responded with different orderings—EPaxos falls to the normal path.

Mencius [116] aims to avoid the client-to-leader overhead by distributing the leadership of SMR log entries across participants. Instead of one agent becoming the leader of all future log entries—as is typical in multi-paxos—the leadership of log entries can be assigned in a round-robin fashion, *i.e.*, agent a is the leader for log entry 0, agent b is the leader for log entry 1, and so on. The benefit of this approach is that a client can send its request to the closest participant to be added to the next available log position where it is a leader. Another benefit of this round-robin assignment of leadership is load balancing.

Leader or majority [13] explores the aspect of mitigating the need to go through the leader for read operations. In a regular RAFT or multi-paxos deployment, the leader with a read lease is the only node that can provide a consistent read. In this work, the authors propose using quorum reads that may bypass the leader when sending read-only requests. This approach leads to better read performance if the round-trip time from the client to a majority of participants is faster than the round-trip time to the leader.

Weighted Replication

Weighted replication [50], [53] is a method of replication that assigns a weight for the vote of each agent. The higher the weight, the higher the influence of the vote. This approach has benefits in geo-replication where an agent can be assigned a weight that corresponds with certain properties such as their priority, distance from others, and level of resilience. Sousa and Bessani [143] explored the use of weighted replication techniques in the context of geo-distributed systems. In particular, their weighing technique allows the utilization of quorums with varying sizes. This flexibility in quorum sizes enables having quorums that are smaller in size and/or separated with shorter wide-area links. These properties improves performance by reducing the latency necessary for coordination.

5.3.2 Hierarchical and Locality-Aware Geo-Replication

This approach for geo-replication redesigns the consensus protocol to leverage the locality of data access. Agents that are spread around the world are grouped into subgroups that coordinate with each other locally. For operations that span multiple groups, coordination between subgroups is performed. This is a hierarchical coordination pattern that enables fast local operations while providing methods for global coordination. It turns out that Flexible Paxos [82] provides the theoretical foundation that allows such hierarchical coordination to be performed. Next are protocols that build on Flexible Paxos to build a hierarchical consensus protocol for geo-replication.

Dynamic Paxos

The system model of Dynamic Paxos (DPaxos) [126] consists of subgroups of agents around the world. Each subgroup represents a unit in the hierarchy. DPaxos utilizes Flexible Paxos by assigning quorums in the following way: (1) Replication quorums: any majority of nodes in a subgroup is a replication quorum. (2) Leader election quorums: any group of nodes that intersect with all replication quorums. This quorum assignment allows a majority of nodes in the subgroup to commit to the

SMR log, which means that they avoid wide-area latency in replication rounds—which are the most frequent. This enables a partition of data to operate closest to its users without incurring wide-area latency. If a partition moves to another location, then a leader election round is performed that would span all subgroups. Although the latency of leader election is high, its cost is amortized by using multi-paxos.

DPaxos allows assigning quorums in a way that allows tolerating subgroup failures where all—or a majority of—nodes in a subgroup experience a failure. This models outages that would impact a geographical location that span agents in a subgroup. This type of outage would halt DPaxos as described above because Leader Election cannot be performed as it has to span all subgroups. To tolerate such outages, DPaxos modifies the assignment of a replication quorum to be a majority in any two subgroups. A leader election quorum is any set of nodes that intersect with all replication quorums. Because each replication quorum must span two subgroups, the leader election quorum can be formed even with the failure of all nodes in a subgroup. This method can be extended to a larger number of subgroup failures (f_g) by ensuring that the replication quorum spans at least $f_g + 1$ subgroups.

DPaxos proposes an extension to Flexible Paxos called *expanding quorums*. The problem that expanding quorums aims to solve is the size of the leader election quorum. With Flexible Paxos, the leader election quorum must intersect all replication quorums, which means that it needs to span most—if not all—subgroups based on the level of subgroup outage tolerance. To make the size of leader election quorums smaller, DPaxos makes the observation that leader election quorums do not need to intersect all replication quorums all the time. Rather, they only need to intersect replication quorums that were previously active, i.e., where acceptors in the replication quorum have received a propose messages. This is because a leader election quorum aims to invalidate prior ballot numbers and learn of previous accepted values. If a replication quorum did not participate in a replication round, this means that there is no need to intersect it in the leader election quorum.

With this observation, DPaxos leader election quorum only needs to ensure intersecting with active replication quorums. Expanding quorums allow a way to find the location of active replication quorums. First,

leader election quorums are assigned to be intersection with each other (note that this is a condition to make leader election quorums intersect with each other which is different from the condition of flexible paxos where a leader election quorum must intersect with all replication quorums.) With this condition, a leader election quorum can be assigned to be any majority of nodes in a majority of subgroups instead of having to span all or most subgroups. These intersected leader election quorums will be used to record what replication quorums are intended to be used by a proposer. A proposer piggybacks the subgroup it intends to use in the prepare message. The acceptor records the intent of every prepare message it acknowledges. Then, the acceptor piggybacks all previous intents that it has received in the prepare-ack message. This means that the proposer, after the leader election round, knows all the replication quorums that were intended to be used (this is because any two leader election quorums intersect.) The proposer then makes a decision to *expand* the leader election quorum if it turns out that there is an active replication quorum that it did not intersect with. The rest of the protocol proceeds in the same way as paxos.

WPaxos

WPaxos [8] is a protocol that aims to utilize Flexible Paxos in geo-replication by grouping nodes into local zones and assigning replication quorums to be within zones. WPaxos proposes the concept of *object stealing* to manage multiple data shards across zones. Each WPaxos instance manages one data shard. Occasionally, a data object in a shard needs to be migrated to another shard, e.g., for load balancing or because the geographic location of the associated used changed. WPaxos utilizes the leader election phase to change the ownership of a data object (or group of data object) from one leader to another—migrating it from one shard to another. Object stealing demonstrates a more practical method for migrating data ownership that avoids the overheads incurred in prior solutions. This is because the ownership change is integrated with the consensus protocol itself, rather than needing to perform external coordination and management to change data ownership.

6

Consensus for Blockchain

To broaden the reach of distributed commit protocols, such as Paxos, we move beyond the limits of the crash-tolerant setting. Such transition is necessary for the practical deployment of consensus protocols in an environment where arbitrary faults may arise. This is primarily to facilitate the emergence of distributed and decentralized applications made possible through blockchain systems.[1] We begin by formalizing the failure models and refining our notion of consensus. We offer a unique unified representation for expressing a wide range of consensus protocols. Most importantly, we aim to present insights into the design of seminal consensus protocols such as Pbft [33]. We offer a fresh perspective on how to navigate and examine the consensus landscape while presenting sufficient insight to reason and differentiate among them. Ultimately, we aim to simplify and make the design of these rather complex and intricate protocols accessible to a wide range of audiences, a stepping stone to further advancement of this field.

[1] We begin by focusing primarily on the permissioned blockchain setting that relies on the notion of identity to register votes to partake in an election in order to reach consensus, referred to as a closed membership. We conclude by shifting focus to permissionless blockchains that support open membership and do not assume the existence of identity (cf. Section 6.5).

We study Pbft as the necessary foundation before exploring speculative, optimistic, and concurrent consensus designs. We further examine the basic mechanism to partially linearize communication patterns in consensus. We conclude by examining the topology of consensus in the context of cross-shard and cross-chain designs. This coverage is visualized in Figure 6.1.

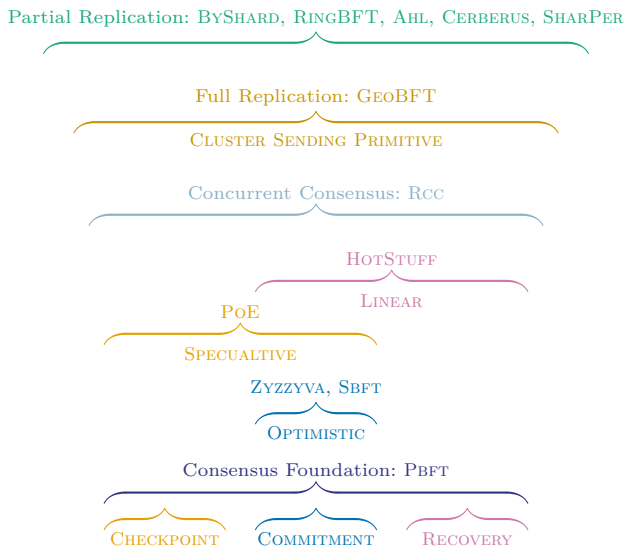


Figure 6.1: A schematic hierarchy of the key concepts to present the foundation of consensus anchored in Pbft, consisting of *commitment*, *recovery*, and *checkpoint* sub-protocols. The consensus landscape is explored through the lens of (1) *speculation*, which delegates the commitment task to the checkpoint routine, (2) *optimism*, which attempts to reduce commitment path, (3) partial consensus *linearization*, which continuously invokes recovery after every commitment. The next tier is focused on *concurrent consensus*, which runs multiple instances of consensus in parallel. The final tier is focused on the topology of consensus through clustered design that gives rise to both full and partial replication schemes.

6.1 Consensus and Failure Model

Consensus, which lies at the heart of blockchain systems, can simply be viewed as an election among a set of agents (*i.e.*, replicas), where in each round of a successful election, the majority of replicas vote to agree

upon a value such as a contract or transaction. To study the problem of consensus in a general setting, which we refer to as *fault-tolerant* consensus, we extend our failure model (cf. 2.1) to allow arbitrary faults of replicas, also known as malicious or Byzantine behavior.[2] In our setting, the faulty behavior of a replica may include both (1) omission, *e.g.*, fail-stop or message loss, and (2) commission, *e.g.*, the equivocation of intent. We begin by first formally describing our setting.

System Model: We model a *system* as a set $\mathfrak{R}$ of *replicas* that process requests[3] from a set of *clients*. We assign each replica $\mathrm{R} \in \mathfrak{R}$ a unique identifier $\mathsf{id}(\mathrm{R})$ with $0 \leq \mathsf{id}(\mathrm{R}) < |\mathfrak{R}|$.[4] We write $\mathcal{F} \subseteq \mathfrak{R}$ to denote the set of *faulty replicas* that can behave in arbitrary ways, possibly coordinated and malicious. We assume that non-faulty replicas behave in accordance to the protocols they are prescribed. Most consensus protocols do not make any assumptions on the clients behavior: all clients can be malicious without affecting the system. We write $\mathbf{n} = |\mathfrak{R}|$, $\mathbf{f} = |\mathcal{F}|$, and $\mathbf{g} = \mathbf{n} - \mathbf{f}$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively. We assume that $\mathbf{n} > 3\mathbf{f}$ and $\mathbf{g} > 2\mathbf{f}$. We further assume that any valid *quorum* of replicas is of size $\mathbf{g} > 2\mathbf{f}$, which is also referred to as *the majority* of replicas.

Given a system of $\mathfrak{R}$ replicas, a single *run* or a single *round* of any *consensus protocol*, as conceptualized in Figure 6.2, should satisfy the following properties:

Definition 6.1. (Primary Consensus Properties)

- *Non-divergence (Safety):* In each round of consensus, only one value is decided by all non-faulty replicas.

[2]We intentionally aim to refrain from the commonly accepted terminology of Byzantine behavior, which carries a negative connotation that categorically, perhaps unintentionally, that demeans an entire civilization that lasted a millennium. Let this be a genuine call to re-consider terminology in computer science by examining its negative connotation and lasting unintended harm. Other examples include the replication model of *master-slave*, operating system concepts such as *kill* a process or *preemptive* scheduling, database concurrency control primitives such as wait-*die* and *wound*-wait, to name a few, all eliciting negative emotions.

[3]We use the term client's request, proposal, or transaction interchangeably.

[4]The existence of identity—and possibly a trusted authority to issue it—is an essential assumption which implies the existence of voting rights to partake in an election. However, having voting rights does not preclude faulty behaviors.

- *Termination (Liveness):* Each non-faulty replica must eventually decide a value.

The safety property is concerned with reaching a correct outcome so that nothing bad ever happens while the liveness property ensures that something good eventually happens, and there is an outcome. A protocol can be trivially safe if it produces no outcome when no value is ever decided upon, but, of course, such a protocol has no practical value. We further introduce additional consensus properties to ensure every round of agreement is not arbitrary. We expect there is an interested party (*i.e.*, client) for each decided value (*i.e.*, validity), who will eventually be informed (*i.e.*, response), and that every client will eventually have the opportunity to propose (*i.e.*, service). We assume that all requests are digitally signed by clients and tamper-proof as such. We further assume that the communication among replicas is authenticated such that a faulty replica cannot impersonate a non-faulty one.

Definition 6.2. (Secondary Consensus Properties)

- *Validity:* Any decided value must have been proposed by a client.
- *Response:* The client of the decided value will eventually be informed.
- *Service:* A well-behaving client will eventually be able to propose a deciding value.

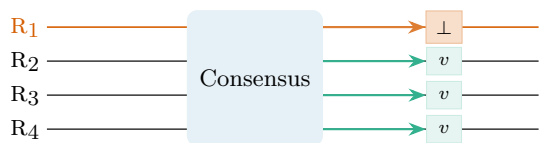


Figure 6.2: A schematic representation of *consensus* among four replicas ($3\mathbf{f}+1$) tolerating $\mathbf{f}$ faults in which R_1 is faulty and is marked as red. Upon completion of *consensus* all non-faulty replicas agree on a common value.

6.2 Consensus Foundation

As a prerequisite to understanding the general problem of consensus, we explore in-depth the seminal fault-tolerant consensus protocol known as PBFT (Practical Byzantine Fault Tolerance) that consists of three main sub-protocols [33], [34].

- A *Commitment Protocol* that enables a well-behaving leader to reach an agreement among the majority of replicas, forming a *commitment quorum*, on a client's proposal via two phases of all-to-all communication (cf. 6.2.2).
- A *Recovery Protocol* that replaces the faulty-leader with the support of the majority of replicas, forming a *recovery quorum*, to preserve any committed proposals and restore liveness via view change and new view phases (cf. 6.2.3).[5]
- A *Checkpoint Protocol* that restores replicas in dark with the support of the majority, forming a *checkpoint quorum*, without relying on a leader to reach a decentralized commitment via checkpoint phase (cf. 6.2.4).

Consensus operates in rounds, and in each *round* a new value based on the client's proposal may be proposed and committed by the *leader* via *the commitment protocol*. Any replica may eventually assume the role of *leader*. Since consensus is driven by a *leader*, intuitively, the entire system can be seen through the lens of the leader, namely, the leader's *view*. Thus, every *leader* represents a *view*. Each newly elected *leader* presents a new *view* of the system. Similarly, *change of leader* implies *change of view*. When *commitment* stalls the view is changed via the *recovery protocol*, but when replicas are left uninformed of the commitment, the view may be restored via the *checkpoint protocol*.

[5] In Section 6.2.3, we further demonstrate that the commitment and recovery paths are almost identical. In commitment, the agreement is reached on the client's proposal whereas, in recovery, the agreement is reached on replicas' prepared proposals.

6.2.1 Preliminary: Crash to Fault Tolerant Transition

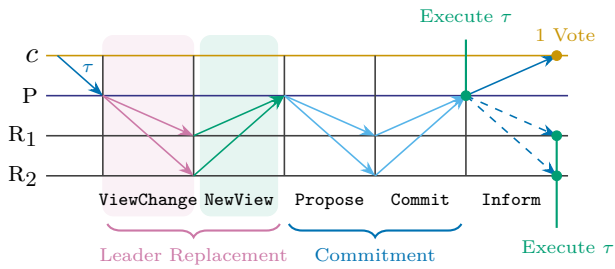
To transition from a crash-tolerant to fault-tolerant setting, we first simplify the standard terminology used to describe Paxos earlier. We adopt the primary-backup model consisting of a *system* of $\mathfrak{N}$ replicas of size $\mathbf{n} > 2\mathbf{f}$ that receive client requests. A replica is elected as the leader, also referred to as the primary or coordinator, akin to classical database agreement protocols such as 2pc and 3pc [141]. We no longer differentiate among roles such as acceptor and learner, and we assume that every replica accepts and learns every decided value.

The basic idea of crash-tolerant consensus protocols such as Paxos is that when a value is proposed (via `Propose`) by an elected leader, then the agreement is reached as soon as the proposed value is endorsed by the majority of replicas (via `Commit`), a reminiscent of two-phase commit protocol, which we refer to as the *commitment protocol* in consensus. These endorsements guarantee that the proposed value is sufficiently propagated in the system, which ensures there is sufficient redundancy to preserve the decided value as long as the majority of replicas are non-faulty. Since the leader is trusted in disseminating the proposal and collecting the votes correctly, the leader is guaranteed to make progress as long as the communication is reliable and the majority of replicas including the leader have not crashed, referred to as a *commitment quorum*. Thus, there is no need for redundant all-to-all communication among replicas to ensure the leader behaves well and proposes the same value to all. Unless a quorum is constructed, namely, the majority are in agreement, no progress is ever made; hence, liveness is lost, yet safety is assured.

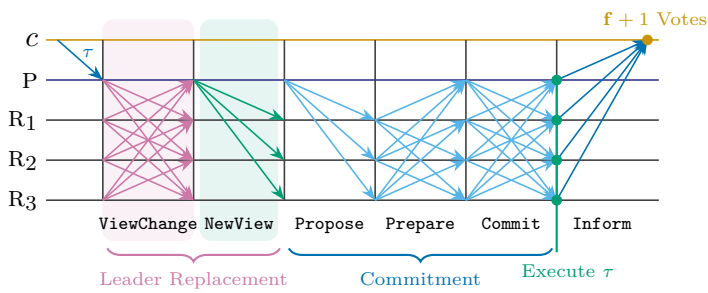
Paxos's recovery protocol for *leader replacement* resembles proposing a value. Any replica may trigger a new election via `ViewChange` message, adopting Pbft style terminology. If the replica is endorsed by the majority via `NewView`, then the replica is elected as the leader.[6] When a replica observes no progress, it often calls for an election to

[6] In Paxos terminology, the `ViewChange` and `NewView` phases are referred to as `prepare` and `prepare-ack`, respectively.

replace the suspected crashed leader to restore liveness.[7] Each replica will independently maintain a local timer to measure the period of no progress in order to decide when a new election is justified. Overall, Paxos can be dismantled into sub-protocols of *commitment* and *leader replacement*. The flow of Paxos is depicted in Figure 6.3a.



(a) Paxos Protocol



(b) Pbft Protocol

Figure 6.3: (a) A schematic representation of Paxos protocol in a setting with $\mathbf{n} > 2\mathbf{f}$ replicas, in which a new primary P is elected via `ViewChange` and `NewView` before proposing client's transaction τ to all replicas via a `Propose` message. Replicas commit to τ via a one-phase of all-to-one message exchange. (b) A schematic representation of Pbft protocol in a setting with $\mathbf{n} > 3\mathbf{f}$ replicas, in which a new primary P is elected via `ViewChange` and `NewView` before proposing client's transaction τ to all replicas via a `Propose` message. Replicas commit to τ via two phases of all-to-all communication, referred to as the *commitment protocol*. The flow of the *leader replacement protocol* is simplified to capture the overall concept.

[7] In general, a replica cannot distinguish between an unreliable network and crashed leader, and in both cases, it will blame the leader and trigger an election.

In contrast, in the fault-tolerant setting, we transition to a *system* of $\mathfrak{R}$ replicas of size $\mathbf{n} > 3\mathbf{f}$ that can handle arbitrary failures. Intuitively, increasing the size of $\mathfrak{R}$ from $\mathbf{n} > 2\mathbf{f}$ to $\mathbf{n} > 3\mathbf{f}$ lies in the fact that $\mathbf{f}$ replicas may behave arbitrarily. Suppose that in the crash-tolerant setting $\mathbf{f}+1$ votes are needed to preserve any decision. Thus, to maintain the same guarantee in the fault-tolerant setting, at least $2\mathbf{f}+1$ votes are required because endorsements from up to $\mathbf{f}$ replicas may be unreliable and revoked arbitrarily [68].

6.2.2 Commitment Protocol: Leader-based Agreement

To dissect the fault-tolerant consensus, we turn to PBFT. The start of PBFT's commitment protocol is traced to when a leader proposes a value (via `Propose`)[8] followed by replicas redundantly exchanging the received value (via `Prepare`) as an oversight to ensure that the leader is behaving as expected. Unlike PAXOS, each replica will independently count endorsements without relying on the leader. Once a replica establishes that the majority has received the same proposed value from the leader, the replica enters the *prepared* state by forming a *prepared quorum*[9]—*i.e.*, *establishing the integrity* of the proposal ensuring no equivocation by the leader. Considering that each replica independently arrives at the *prepared* state, a *prepared* replica is unable to infer whether other replicas have also prepared or not, and it is possible that only one non-faulty replica reaches the prepared state due to message loss and unreliable network.

Thus, a *prepared* replica must exchange its commitment to the proposed value (via `Commit`) with all replicas. Once a replica receives commitment from a majority of replicas, *forming a commit quorum*, it can infer that a majority of non-faulty replicas have prepared, and an agreement is reached—*i.e.*, *establishing the preservation* of the proposal

[8] In the original PBFT paper, the propose phase is referred to as pre-prepare.

[9] In PAXOS, as soon as a replica receives a proposal from the leader, it transitions to the *prepared* state as it can be certain that no other replicas would be receiving a conflicting proposal from the same leader, eliminating the need to form prepared quorum.

across views.[10] As long as a single non-faulty replica reaches the *committed* state, it is guaranteed that the committed value will always be recoverable (*i.e.*, preserved) because there will always be at least one non-faulty prepared replica in every possible quorum. On the contrary, if a non-faulty replica assumes consensus after reaching the *prepared* state and informs the client,[11] then the preservation of the promise to the client is not guaranteed. Because shortly after, the recovery procedure may be triggered, and it is possible that the recovery quorum does not include the only non-faulty replica that was prepared; thus, the commitment to the client could be overwritten. The flow of PBFT is captured in Figure 6.3b.

A closer look at PBFT reveals that only the proposal phase is leader-based while the remaining phases are, in fact, leader-less, exhibiting a decentralized agreement and convergence. We can conceptualize the *commitment protocol* consisting of two main stages, (1) *proposing* a value and (2) *agreeing* upon the proposed values. Having the proposal stage driven by a leader substantially simplifies the design of the consensus protocol, mimicking a centralized behavior by allowing a single replica to choose the next proposal. The agreement stage can be carried out in a decentralized manner in which all replicas directly exchange their endorsements with everyone.

More formally, the notion of the agreement itself is defined by the formation of a quorum, that is $2\mathbf{f}+1$ replicas endorsing the same value out of $3\mathbf{f}+1$ replicas. Alternatively, the size of the quorum can be defined as the number of non-faulty replicas. The importance of quorum lies in the *set-intersection property* to prove that all possible pairs of quorums are overlapping, and the overlap contains at least one non-faulty replica. Therefore, through this common non-faulty replica, it is guaranteed that any two possible quorums will never reach conflicting decisions because

[10] Theoretically, the completion of consensus is established only when all non-faulty replicas (not just the majority) have committed to the proposed value. Ensuring all non-faulty replicas are up-to-date can be achieved through a decentralized checkpoint process that does not rely on the leader.

[11] It is important to note that in PBFT, it is sufficient to have a single non-faulty replica informing the client about the commitment, which is the same as in PAXOS. Even though in PBFT, the client expects $\mathbf{f}+1$ endorsements, $\mathbf{f}$ of which may be attributed to faulty replicas who may revoke their support subsequently.

no non-faulty replica will ever endorse conflicting proposals. For the formal analysis of PBFT and its correctness proof, we refer interested readers to [64].

What is of crucial importance in the design of PBFT is that progress is made (*i.e.*, liveness) only when a quorum is established. By construction, any progress made upon the formation of a quorum is always correct because all possible quorums overlap through a non-faulty replica that prevents divergence (*i.e.*, safety).

6.2.3 Recovery Protocol: Leader Replacement

But what if no *commitment quorum* is established to ensure progress? This is the central question that must be tackled in the recovery mechanism of any consensus protocol. But first, the lack of progress itself must be detected reliably, which in itself requires the formation of what we refer to as the *recovery quorum*. Thus, it becomes evident that consensus protocol, such as PBFT and all of its derivatives, will stall indefinitely unless a quorum is formed; said differently, *no step is ever taken in* PBFT *unless there is a quorum that endorses this step*. This fundamental principle that necessitates the support of a quorum holds true for both commitment and recovery flow of PBFT.

Therefore, if for any reason a replica in PBFT detects a lack of progress independently through its local timer or detects malicious behavior of the primary,[12] it will aim to replace the leader, *i.e.*, changing the view. Note that a view change does not imply faulty primary behavior nor precludes it, as network unreliability may also impede progress. This leads to an *indistinguishability dilemma* such that lack of progress may be due to the network failing to deliver the message or the primary failing to send the message. But in either case, the protocol finds a resolute in replacing the leader indefinitely until both the network and primary are sufficiently well-behaving such that a quorum can be formed.

The recovery routine consists of three main stages. *First*, the detection is triggered independently via `ViewChange` phase, which needs the support of the majority to form a *well-formed* recovery quorum before proceeding any further. *Second*, the recovery quorum must agree

[12] A faulty primary may propose two different values in the same round.

upon who the next leader is; this is a subtle yet critical synchronization step, which is arguably the most vital and overlooked mechanism of PBFT protocol. This synchronization must tolerate and recover from repeated arbitrary failures during the recovery itself. *Third*, the new view must be established based on the information provided by all non-faulty replicas as part of the `ViewChange` message, which includes all proposals proposed and prepared for every round since the last stable checkpoint (cf. 6.2.4). The new view is delivered via `NewView` message to all replicas. Considering the `ViewChange` messages were exchanged among all replicas,[13] all replicas can independently verify the validity of the new view proposed by the new leader. The flow of the view change protocol is illustrated in Figure 6.4.

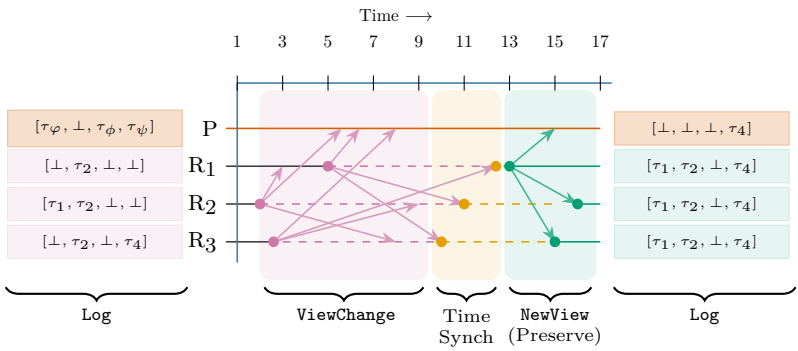


Figure 6.4: Failure Recovery: A schematic representation of the PBFT's leader replacement protocol, also referred to as the view change protocol. All replicas detect the primary P failure and initiate leader replacement via `ViewChange` message. Between the time T_9 to T_{13}, all replicas observe the recovery quorum formation of {R1, R2, R3}. At time T_{13}, R1 is elected as the new leader and broadcasts the `NewView` message consisting of $[\tau_1, \tau_2, \perp, \tau_4]$. Via `NewView`, all logs are eventually brought up-to-date through consensus by preserving any probable committed proposals.

Further note that neither the new view nor the recovery quorum is necessarily unique; therefore, the primary may choose any well-formed quorum to construct a new view. The main requirement of the new view is to preserve all committed proposals logged by each replica. For

[13] A quadratic message complexity is necessary to exchange view change messages and reach synchronization, which cannot be reduced through threshold signature employed in a protocol such as [54], [62], [157].

example, suppose a value was proposed in a round but never prepared or committed. In that case, the primary has the liberty to either preserve one of the possible uncommitted values or opt out for a no-op ($\perp$) value instead. Given a recovery quorum, for every round ρ_i, if the value τ was prepared by at least one non-faulty replica—which can be demonstrated through a prepared certificate digitally signed by a quorum or if there is a quorum of non-conflicting prepared claims[14] within the same view, where there is at least one non-faulty replica claiming τ was proposed in the same or later view,—then τ must be preserved in the round ρ_i of the new view. But with inadequate prepared support, no-op is assigned to ρ_i. After the view is changed, the logs (*i.e.*, ledger) of all participating non-faulty replicas are brought up-to-date by having the primary to guarantee the uniqueness of the new view, *i.e.*, ensuring there exists a quorum that agrees on the same new view. The view change further serves as the means to recover replicas in dark.

Considering the recovery quorum may not be unique because not every round has been committed since the last stable checkpoint, then a malicious primary may not share the same recovery quorum with all replicas. To guard against such attacks, in PBFT, post view-change, the new primary must run a consensus on every round ρ_i since the last checkpoint for which there is a prepared proposal or no-op in the new view to ensure that a quorum will observe a unique new view. Consequently, the new view may be seen as a set of promised proposals, which primary must propose through consensus to ensure uniqueness after broadcasting the new view. If a committed certificate exists for any round ρ_i in the new view, no additional round of consensus is needed. For any replica that did not receive a committed certificate for round ρ_i, it would expect the primary to propose τ or $\perp$ for round ρ_i. Otherwise,

[14] In order to construct a non-conflicting quorum, *i.e.*, a well-formed quorum, the new leader may be forced to wait for all non-faulty replicas to participate. This further implies that every possible recovery quorum may not be well-formed because faulty replicas may provide conflicting information (in absence of a certificate) forcing the leader to stall with a chance of instigating another view change after a single non-faulty replica times out [64]. Forming a non-conflicting quorum of prepared claim is only possible if a majority of non-faulty replicas either support or have no objection to the claim; thus, no other conflicting proposal could have been committed, *i.e.*, a safe claim.

the replica would conclude that the new primary is misbehaving and would trigger another primary replacement.

Similar to the commitment flow of the protocol, we observe that the leader replacement of PBFT is only partially leader-based. The detection mechanism is leader-less, but once the majority agrees upon the new view, then constructing and propagating the new view is carried out by the new leader, making the second part leader-based.

We expand our investigation to consider what happens when we face failures during the recovery. For example, the new leader may behave maliciously by not sending the new view message or proposing a conflicting or inconsistent new view. The network may also drop or delay messages indefinitely, exhibiting an unreliable behavior. The recovery is entirely based on the timeout period, so now the central question that arises is what happens if a replica times out again after issuing a view change message? Or maybe a replica should wait indefinitely once it raises a view change?

Essentially the view change serves as a lockstep to ensure the network and the majority of all replicas are sufficiently synchronized. It is centered around the basic principle of *the formation of recovery quorum* to achieve this subtle time synchronization intricacy.

1. *What would happen if a replica times out after sending its view change message but before observing a recovery quorum?*

2. *What would happen if a replica times out after observing the recovery quorum?*

The solution is fairly simple yet powerful. As long as recovery quorum is not formed, *the timed out replica will indefinitely re-transmit its view change message*, *i.e.*, *as if time stops*. Once the recovery quorum is formed, then *a replica will issue a new view change for the next leader*, *i.e.*, *time is advanced*. Thus, all non-faulty replicas synchronize on the single event of quorum formation to agree upon who the next leader is in order to synchronize their time and state. An example of repeated failures that is resolved by time synchronization is presented in Figure 6.5.

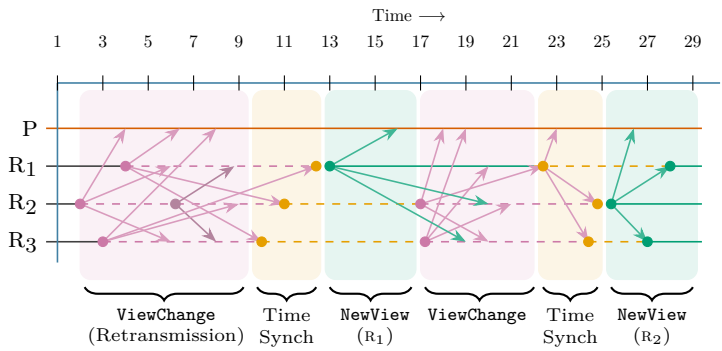
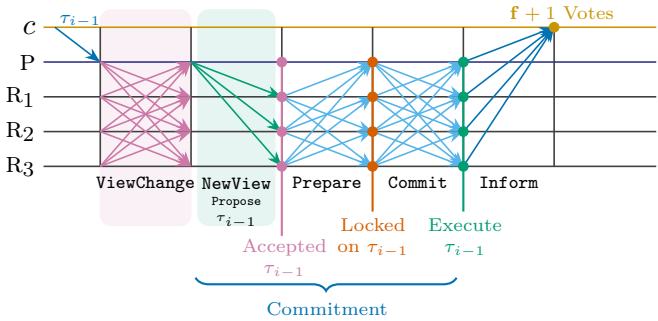


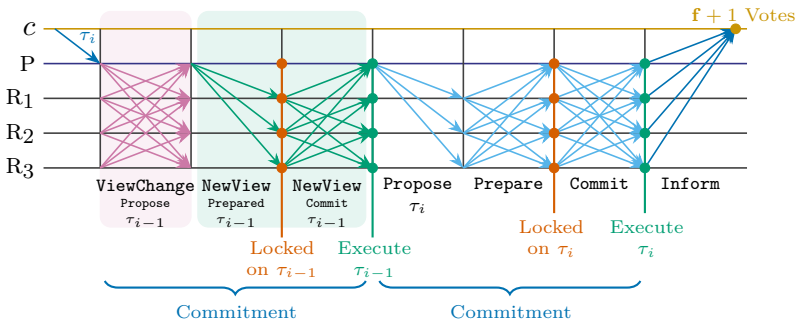
Figure 6.5: Repeated Failure Recovery: A schematic representation of the PBFT view change protocol that tolerates repeated failures. All replicas detect the primary P failure and initiate leader replacement via `ViewChange` message. After sending its view change message, R2 timer expires at time T_6 before observing the recovery quorum; therefore, R2 re-transmits its `ViewChange` message. Between the time T_9 to T_{13}, all replicas observe the recovery quorum formation of {R1, R2, R3}. At the time T_{13}, R1 is elected as the new leader and broadcasts the `NewView` message. However, both R2 and R3 time out at T_{17} before receiving `NewView` from R1. Thus, both R2 and R3 concludes the failure of R1 and initiates another view change to elect R2 as the next leader. Between the time T_{22} to T_{25}, all replicas observe the formation of the recovery quorum. At the time T_{25}, R2 is elected as the new leader and broadcasts the `NewView` message successfully.

Considering that PBFT operates in partial synchrony, each replica relies on correctly guessing the unknown but the bounded network delay to set its local timer. If the chosen time-out period is too small, replicas may continuously trigger view change, and no progress is made. If the time out is too large, then the malicious primary and replicas may artificially delay sending their messages, reducing the overall progress proportional to the presumed network delay. Therefore, the replicas can start with a reasonably small timeout based on the average expected delay of the network. Still, if an inadequate length is chosen, the time-out period can increase exponentially during view change, taking an effect in all subsequent rounds.

To summarize, we emphasize that PBFT *implicitly makes progress (to ensure liveness) in lockstep where a quorum (to ensure safety) is always necessary to lock and unlock any steps.* When a replica reaches the prepared state for the proposal τ after observing a prepared quorum for



(a) PBFT Recovery Protocol



(b) Complete PBFT Protocol with Optimized Recovery

Figure 6.6: (a) A schematic representation of PBFT recovery as lockstep flow. A new primary P is elected via `ViewChange` and `NewView`. As part of `ViewChange`, all well-behaving replicas broadcast τ_{i-1} that was previously prepared at the round ρ_{i-1}. As part of `NewView`, the primary constructs a recovery quorum that includes τ_{i-1} at ρ_{i-1}, which indirectly serves as a proposal for τ_{i-1} at ρ_{i-1}. Replicas accepts τ_{i-1} as a valid proposal to be recovered at ρ_{i-1}. Once the new view deemed valid, then replicas commit to τ_{i-1} via two phases of all-to-all message exchange. The primary may propose τ_i at ρ_i when broadcasting the τ_{i-1} proposal for the round ρ_{i-1}.
(b) A schematic representation of the complete PBFT protocol as lockstep flow that entails both the optimized recovery and commitment flow. As part of `ViewChange`, all well-behaving replicas broadcast τ_{i-1} that was previously prepared at the round ρ_{i-1}, which indirectly serves as a decentralized proposal for τ_{i-1} at ρ_{i-1}. As part of `NewView`, the primary constructs a recovery quorum that includes τ_{i-1}, which may indirectly serve as a prepared quorum for τ_{i-1} at ρ_{i-1}. Replicas accept and lock τ_{i-1}. Replicas commit to the new view of τ_{i-1} via one phase of all-to-all message exchange. Once the new view is committed, then the primary P follows the commitment flow of the protocol to propose the client's transaction τ at the round ρ_i to all replicas via `Propose` message. Replicas commit to τ via two phases of all-to-all message exchange.

τ, it is implicitly locked on the prepared state of τ. Subsequently, when a replica reaches the commit step after observing a commitment quorum, it commits τ. A commit of τ is guaranteed never to be overwritten and will endure any future recovery invocations. However, the prepared state of τ may be overwritten, implicitly releasing the prepared lock on τ during the view change if the presented recovery quorum does not include τ with sufficient support. Furthermore, any proposal τ included in the new view that is supported by a recovery quorum can be viewed as an implicit locked prepared state for τ, which means unless the new leader commits it, the proposal τ may be overwritten in the subsequent views. However, as soon as any non-faulty replica commits τ, it will indefinitely re-appear in every possible recovery quorum, so it will never be overwritten. The lockstep flow of the PBFT protocol along with the commitment of the new view is presented in Figure 6.6.

Looking deeper into the recovery protocol and examining the required phases to establish a unique new view, we can observe that the commitment and recovery are nearly identical as annotated in Figure 6.6. Both commitment and recovery involve `Propose`, `Prepare`, and `Commit` phases. Their distinguishing factor is as follows. Through the commitment protocol, the agreement is reached on the client's proposal in the current view, whereas through the recovery protocol, the agreement is reached on replicas' prepared proposals, namely, what clients had previously proposed in earlier views. Furthermore, the recovery and commitment can further be optimized by allowing the new primary to restore the previously prepared proposal (e. g., ρ_{i-1} in Figure 6.6) and the new proposal (e. g., ρ_i) at the same time as part of its `ViewChange` message. Therefore, reducing the number of phases by collapsing the recovery and commitment as shown in Figure 6.7.

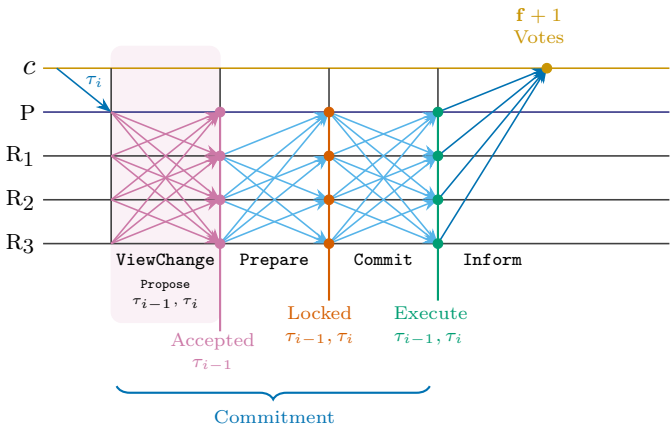


Figure 6.7: A schematic representation of the complete PBFT protocol as lockstep flow, in which τ_{i-1} was prepared at ρ_{i-1} while τ_i is being proposed in the round ρ_i. The representation is further simplified and optimized by allowing the new primary to include the proposal for the new round ρ_i as part of its `ViewChange` along with what it was prepared previously at the round ρ_{i-1}, essentially eliminating the need to have an explicit `NewView` phase as required in Figure 6.6. In a sense, offering an optimistic mechanism that assumes that the new non-faulty leader is aware of what was proposed previously while allowing all replicas to independently construct the same new view confirmed via `Prepare` phase.

6.2.4 Checkpoint Protocol: Decentralized Commitment

To avoid re-committing the entire history repeatedly, PBFT periodically executes a decentralized checkpoint protocol after a fixed, but configurable number of rounds, referred to as the checkpoint window with size w that ranges between the low and high watermarks, l and h, respectively. After every w rounds, all replicas expect to checkpoint all preceding rounds smaller than h. Unless the checkpoint is successful, which may fail because the primary has kept $\mathbf{f}$ replicas in dark or the network is unreliable, replicas will stop accepting any new proposal forcing the replacement of the primary.

After every w round, replicas broadcast a single checkpoint message with a digest that represents the decision of every round since the last stable checkpoint. Once a replica receives $2\mathbf{f} + 1$ matching checkpoint messages, *i.e.*, a *checkpoint quorum*, this replica marks it as a stable checkpoint. Any replica in the dark that receives $\mathbf{f} + 1$ matching check-

point messages can be assured that all the decisions are final because at least one non-faulty replica has committed every single round. Therefore, checkpoint serves as the means to recover the replicas in dark.[15] In turn, the recovered replicas may further contribute to the formation of the checkpoint quorum to reach a new stable checkpoint. The flow of the checkpoint protocol is presented in Figure 6.8.

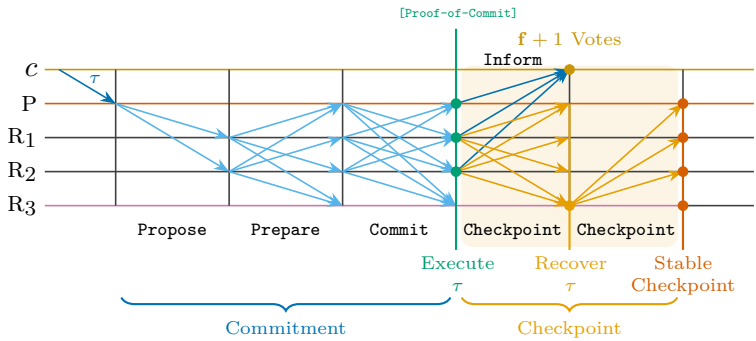


Figure 6.8: A schematic representation of PBFT checkpoint protocol, in which the malicious primary P does not send `Propose` message to R_3, keeping R_3 in dark. The remaining replicas commit to τ via two phases of all-to-all message exchange. The non-faulty replicas R_1 and R_2 initiate `Checkpoint` phase, forming a weak quorum of size $\mathbf{f}+1$ sufficient to recover R_3. Subsequently, R_3 broadcasts its own `Checkpoint` message to form the checkpoint quorum and establishing the stable checkpoint.

The key to the power of checkpoint lies in its decentralized design that avoids the need to rely on any leader to drive progress, but if no progress is made, it conveniently falls back on the view change. In the commitment flow of PBFT, the leader may keep a set of replicas in dark without violating the consensus liveness. In contrast, during the checkpoint, as long as the network is reliable, all non-faulty replicas will broadcast their checkpoint state that allows recovering any replicas kept in dark. Any proposal τ_i at round ρ_i that is supported by a checkpoint quorum will be part of the stable checkpoint and marked as permanently committed. As a result, the view change protocol itself can further be

[15] It will be the replica's responsibility to reach out to at least $\mathbf{f}+1$ replicas in order to determine the actual client's proposals that were committed in each checkpointed round if not known already.

optimized by only constructing a recovery quorum since the last stable checkpoint.

6.2.5 Optimization: Amortized Pipeline Design

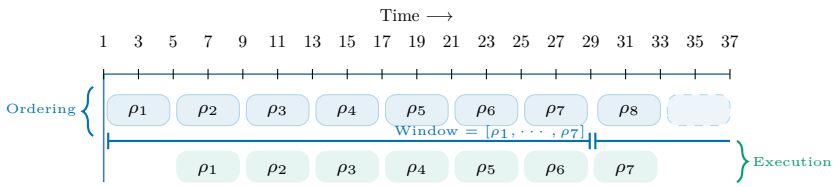
There are basic yet essential optimization applicable to PBFT such as (1) batching proposals, and (2) out-of-order processing of proposals [33], [34], [64], [69]. The former is relatively a standard optimization technique that is to run consensus on a batch of proposals instead of running consensus on one proposal at a time. Thus, amortizing the cost of consensus over the size of the batch. The second optimization is rather involved and may influence how the execution and recovery are carried out.

When there is no out-of-order processing, we only start consensus on a new proposal (or a batch of them) once consensus for the current proposal is completed, *i.e.*, sequential consensus. With out-of-order processing disabled, the network and the entire pipeline, especially the backup replicas, are often underutilized. By enabling it, the primary does not have to wait to complete the first round before starting subsequent rounds. In general, replicas are not required to prepare or commit the proposal τ_i before endorsing the proposal τ_j, where i and j corresponds to the rounds ρ_i and ρ_j and $i < j$. This allows the primary to issue as many proposals as possible in order to fully saturate its pipeline. Additionally, both the primary and replicas can respond to messages out of order, while tolerating any arbitrary re-ordering of messages by the network as well.

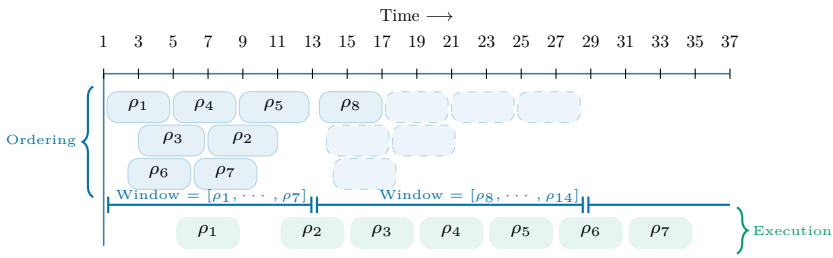
Noteworthy, the actual execution will always be serialized, so in a sense, we are decoupling ordering, *out-of-order commitment*, from the serial execution on the already ordered proposals. This optimization introduces a new kind of attack, in which the primary assigns and orders proposals for every round but skips the first round. Even though the ordering is completed successfully for all proposals, the entire execution is stalled because execution, unlike ordering, must be done serially and requires the commitment of the first round before executing the second proposal and so on.

PBFT handles such attacks by relying on the notion of low- and high-watermark window (*e.g.*, l and h) such that all proposals between the rounds ρ_l and ρ_h must be completed (*i.e.*, checkpointed) before the

primary can advance beyond the round ρ_h. Consequently, all replicas will stop accepting new proposals unless all rounds smaller than and equal to the high watermark are committed. When replicas stop accepting new proposals, clients will inevitably complain, and eventually, the primary is voted out, and the view is changed. The consensus flow of ordering and execution are visualized in Figure 6.9.



(a) In-Order Processing



(b) Out-of-Order Processing

Figure 6.9: (a) A schematic representation of running consensus sequentially, *i.e.*, in-order processing, such that consensus on the round ρ_j does not start until the round ρ_i is completed, where $i \leq j$. Each blue box represents a consensus step for the round ρ_i while each green box represents the execution step of the proposal decided in the round ρ_i. (b) A schematic representation of running consensus out-of-order such that the primary may initiate consensus on the round ρ_j before completion of previous rounds in its low and high watermark window, which is set to 7. Consensus is carried out-of-order while the execution is strictly processed in round order.

6.3 Consensus Landscape

In this section, we explore how the design of PBFT can be advanced in a principle way as we survey four essential design patterns: speculation, optimism, concurrency, and linearization.

6.3.1 Speculative Consensus: Delayed Commitment

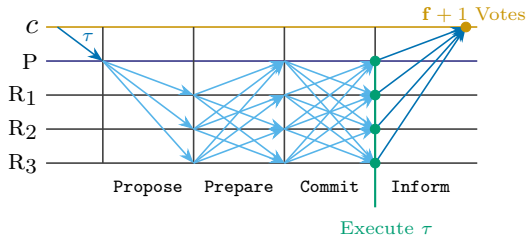
A natural optimization question arises as to whether all phases of PBFT are necessary or whether we can reduce them. *We conjecture that both phases are necessary. If we drop the commit phase, liveness is potentially affected, and if both prepare and commit phases are dropped, safety is affected.* However, this does not rule out creative ways to delay or amortize them speculatively.[16]

Let us re-examine PBFT, as shown in Figure 6.10a. In the prepare phase, we establish that the primary proposed the same value to a quorum (cf. *prepared quorum*), and in the commit phase, we establish that an entire quorum is aware that the primary proposed the same value (cf. *commitment quorum)*. In other words, from the perspective of a replica after the prepare phase, this replica is only aware of its own prepared state, *i.e.*, that the primary behaved and proposed the same value to a quorum. But after the commit phase, a replica is also aware of the prepared state of an entire commitment quorum, *i.e.*, knows a quorum has prepared. The basic principle is to ensure sufficient redundancy is propagated, and there is at least one non-faulty observer that can attest to the existence of a commitment quorum since any recovery quorum would overlap with the commitment quorum, and the proposal will be preserved upon recovery.

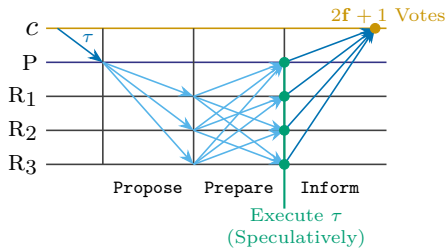
In PBFT, the observer is assumed to be a replica, a sufficient but not a necessary condition. This is the precise intuition behind PROOF-OF-EXECUTION (POE) that allows the client to take on the role of the observer [62], [74].[17] As long as the client observes a *commitment quorum*, the preservation of its proposal will be certain to survive any interruption after recovery. Interestingly, replicas in POE are only aware of their own prepared state and unaware of the commitment state. Any replica that reaches the prepared state will speculatively execute transactions serially and inform the client. Subsequently, the client expects to receive matching speculative execution from a quorum to construct a *proof-of-execution*.

[16] An alternative approach to reduce the number of phases is to increase redundancy [11], [61], [96].

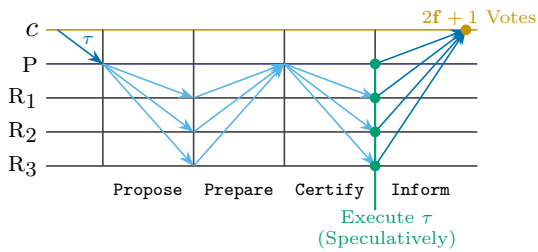
[17] A similar observation was hinted at in the original PBFT paper described as tentative execution along with an abort mechanism of reverting back to the last checkpoint in presence of failures [33].



(a) PBFT Protocol



(b) PoE Protocol



(c) Linear PoE Protocol

Figure 6.10: (a) A schematic representation of the PBFT protocol, in which the primary proposes the client's transaction τ to all replicas via a `Propose` message. Replicas commit to τ via two phases of all-to-all message exchange. The client expects $\mathbf{f}+1$ matching responses to conclude the commitment. (b) A schematic representation of the PoE protocol, in which the primary proposes the client's transaction τ to all replicas via a `Propose` message. Replicas commit to τ via one speculative phase of all-to-all message exchange via `Prepare`. However, PoE expects the client to collect $2\mathbf{f}+1$ matching responses before concluding the commitment. (c) A schematic representation of the Linear PoE protocol, in which the single phase of all-to-all message exchange is broken down into two linear phase of `Prepare` and `Certify`, delegating the aggregation of votes to the primary using threshold signature.

In the absence of any commitment quorum before execution, it is plausible that during recovery, a replica in PoE discovers that its speculatively executed proposal was not selected in the recovery quorum, forcing the replica to roll back contrary to Pbft's design. The proposal can only be excluded from a recovery quorum if less than a quorum of replicas prepared it, implying that it could not have been committed and the client could not have observed a commitment quorum. So excluding it does not violate safety. The flow PoE as shown in Figure 6.10b that can further be transformed into a linear protocol by applying the threshold signature optimization as illustrated in Figure 6.10c. Basically, the threshold signature can linearize any all-to-all communication into two linear phases of all-to-one and one-to-all, where one replica, often the primary, acts as an aggregator to collect individual signatures to construct a certificate.

Although PoE reaches commitment safely after a client observes a commitment quorum, it may temporarily be subject to the client's liveness break, but not a liveness break for consensus. PoE's suffers from what we refer to as a *commitment imbalance*.

In Pbft, we expect that *at least the majority of non-faulty replicas* ($\mathbf{f}+1$) contributing to both the prepared and commitment quorums, *i.e.*, *the minimum commitment necessity*, allowing up to $\mathbf{f}$ faulty replicas in the quorum formation. But, more importantly, in Pbft, we expect that *at most the majority of non-faulty replicas* have to inform the client, *i.e.*, *the maximum informed necessity*. Thus, by design, *the minimum commitment necessity* is always equal or greater than *the maximum informant necessity*, in other words, commitment implies a successful informant when the communication is reliable.[18]

Similarly, in PoE, we also expect that *at least the majority of non-faulty replicas* ($\mathbf{f}+1$) to contribute to the prepared quorum, *i.e.*, *the minimum commitment necessity*, allowing up to $\mathbf{f}$ faulty replicas in the quorum formation. In contrast, PoE, expects that *all non-faulty replicas* ($2\mathbf{f}+1$) to inform the client, *the maximum informed necessity*, because unavoidably the $\mathbf{f}$ faulty replicas who may have assisted in

[18]Note, if the majority of non-faulty fail to reach commitment, then the recovery protocol is invoked, which will eventually restore the liveness.

the prepared quorum formation may not comply. Unfortunately, *the minimum commitment necessity* is no longer guaranteed to be equal or greater than *the maximum informed necessity*; hence, speculative commitment does not imply a successful informant even when the communication is reliable. This subtle balance is violated in PoE and may affect any protocol that forces a client to collect a stronger quorum of size $2\mathbf{f} + 1$ instead of $\mathbf{f} + 1$.

Instead of the client, if the commitment observer was a replica, as in Pbft, then when no non-faulty replica observes a commitment quorum, the recovery will eventually be invoked. But since only the client may observe the commitment quorum in PoE, it will not have sufficient power to unilaterally trigger the recovery as long as the majority of replicas have been prepared. Aligned with Pbft, no single participant will ever have the power to convince the majority to initiate the recovery as at least $\mathbf{f} + 1$ endorsements are needed unless verifiable proof of misbehavior can be presented. Thus, in PoE, the response to the client may stall until the outcome of speculation is consolidated among non-faulty replicas, which can be obtained during the checkpoint phase. But as long as the majority of replicas have prepared the client's proposal, then consensus remains both live and safe while the client's liveness break remains undetected. If the majority of non-faulty replicas did not reach the prepared state, then non-faulty replicas will support the client's objection that no commitment quorum was formed.

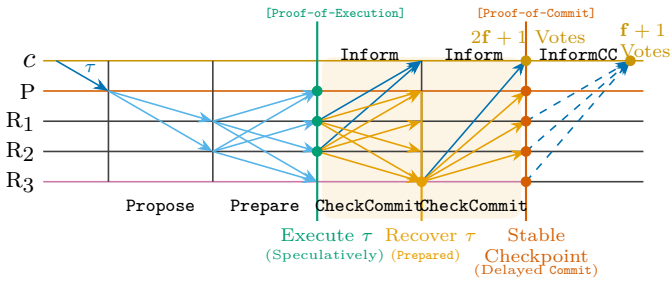
Instigating such a client liveness attack is rather simple. Let's consider we have three sets of replicas, set $\mathcal{F}$ to present $\mathbf{f}$ faulty replicas, $\mathcal{D}$ to present $\mathbf{f}$ non-faulty replicas kept in dark, and $\mathcal{G}$ to present $\mathbf{f} + 1$ non-faulty replicas. The malicious primary keeps $\mathcal{D}$ in the dark, and given $\mathcal{D}$ is smaller than $\mathbf{f} + 1$, $\mathcal{D}$ is unable to trigger view change. The malicious primary orders the proposal through a single phase of consensus, in which both $\mathcal{F}$ and $\mathcal{G}$ behave to construct the prepared quorum. Hence, the consensus remains live and safe. If $\mathcal{F}$ does not comply with forming the prepared quorum, then $\mathcal{D}$ and $\mathcal{G}$ together can trigger view change. However, if $\mathcal{F}$ behaves sufficiently well to advance consensus, but $\mathcal{F}$ does not inform the client, then the client is unable to form the commit quorum. Now suppose the client complains and only the set $\mathcal{D}$ supports the client's claim. This will be insufficient to cause the view change. As a result, the client will stall.

Due to the elimination of the commit phase, clearly, there is a potential client liveness attack in PoE that can only be addressed through the checkpoint phase, which serves as a decentralized delayed commit phase. We can perceive PoE as the principle approach to delay (but not eliminate) the commit phase to post execution. Therefore, in PoE, the commit phase is optimistically removed from the consensus' critical path in order to inform the client with fewer phases when there is a well-behaving quorum of replicas. To circumvent the commitment imbalance, PoE introduces a sub-routine, referred to CheckCommit protocol, that not only restores the client's liveness but also serves as a checkpoint mechanism.

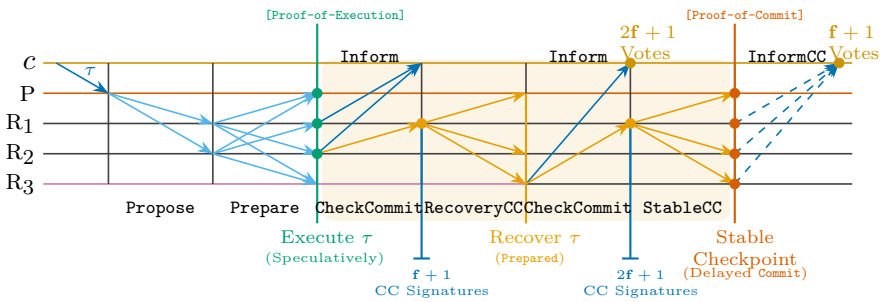
The basic flow of CheckCommit protocol is initiated via the `CheckCommit` phase as illustrated in Figure 6.11a. Similar to Pbft's checkpoint, a non-faulty replica periodically invokes `CheckCommit` phase after every k^{th} rounds. For example, at round ρ_k, a non-faulty replica broadcasts `CheckCommit` if it has already reached prepared state for the last k rounds. A prepared replica that obtains $2\mathbf{f}+1$ `CheckCommit` messages for round ρ_k will consider it as a stable checkpoint. However, if a replica never reached the prepared state for the round ρ_k, but it receives $\mathbf{f}+1$ `CheckCommit` messages it will consider the last k^{th} rounds as prepared. Furthermore, it will issue its own `CheckCommit` message to assist non-faulty replicas in reaching stable checkpoint state in a decentralized manner without relying on any primary.

Any replica that reaches the prepared state will also send `Inform` message to the client as well. This will allow the client to collect its necessary $2\mathbf{f}+1$ endorsements. Alternatively, when a replica reaches the stable checkpoint, it may also send `InformCC` message to the client, conveying to the client that not only was its proposal prepared, but it was also committed. As a result, it would be sufficient for the client to collect $\mathbf{f}+1$ matching `InformCC` instead of the $2\mathbf{f}+1$ `Inform` messages that convey only the preparedness. If the non-faulty replicas are unable to reach the stable checkpoint state after every k rounds, then they simply blame the primary and fall back to view change to restore the state.

In contrast to the commitment flow of PoE that permits out-of-order processing, the CheckCommit protocol adopts in-order processing of



(a) CheckCommit Protocol in PoE



(b) Linear CheckCommit Protocol in PoE

Figure 6.11: (a) A schematic representation of PoE's CheckCommit protocol. The malicious primary P does not send `Propose` message to R3, keeping R3 in dark. The remaining replicas commit to τ via one speculative phase of all-to-all message exchange via `Prepare`; however, the client is unable to form a commitment quorum consisting of $2\mathbf{f}+1$ votes, resulting in an undetectable client liveness attack. The non-faulty replicas R1 and R2 broadcasts `CheckCommit` message, forming a weak quorum of size $\mathbf{f}+1$ sufficient to recover R3. Subsequently, R3 informs the client to restore client liveness and broadcasts its own `CheckCommit` message to form a commitment quorum. Upon reaching the stable checkpoint, replicas (if needed) may send `InformCC` to the client implying the replica has `Proof-of-Commit`; hence, the client would only require to receive $\mathbf{f}+1$ `InformCC` matching votes. (b) A schematic representation of the linear CheckCommit protocol. Through a round-robin assignment, each replica is designated as a vote aggregator such that R_i is assigned to round ρ_j, where $i = j \bmod n$. Suppose R1 is assigned as the aggregator. The non-faulty replicas R2 signs `CheckCommit` message with both weak and strong threshold signatures. R1 using its own vote can form a weak quorum of size $\mathbf{f}+1$ sufficient to recover R3 by constructing `RecoveryCC` that is supported by $\mathbf{f}+1$ signatures. Subsequently, R3 sends `CheckCommit` message to R1 enabling R1 to construct and broadcast a commitment quorum via `StableCC` supported by $2\mathbf{f}+1$ threshold signatures.

`CheckCommit` phase, which lies outside the critical path of consensus. Furthermore, the choice to adopt in-order processing is not necessary to ensure safety or restore client liveness. It is only intended to allow PoE reach the stable checkpoint after every k rounds of consensus. This would reduce the size of the view change message (as adopted by Pbft) by only including the prepared state for every round since the last stable checkpoint. Suppose the `CheckCommit` is not processed in-order. As a result, ρ_k may be check committed successfully while the fate of the ρ_i (where $i < k$) has yet to be determined, implying that all non-faulty replicas need to provide information about ρ_i as part of their view change message. Moreover, considering that replicas are unaware of each other's prepared states, then they would be forced to provide information for all rounds if CheckCommit was not processed in-order.

The CheckCommit protocol is further optimized into a general linear protocol, as presented in Figure 6.11b, that can form a new foundation to design more efficient checkpoint protocols, also applicable to Pbft. Every round of consensus (or every k^{th} round) is assigned to a replica as `CheckCommit` vote aggregator in a round-robin fashion, *e.g.*, R_i is assigned to the round ρ_j, where $i = j \bmod n$. The assigned replica aggregates all `CheckCommit` votes to construct a threshold signature certificate and broadcasts it to all replicas. Essentially transforming a single phase of all-to-all communication into two phases of all-to-one and one-to-all communication. Considering that up to $\mathbf{f}$ non-faulty replicas may be in dark, then a non-faulty aggregator may only receive $\mathbf{f}+1$ `CheckCommit` votes, including its own vote. Therefore, the aggregator aims to construct both weak and strong certificates, which require $\mathbf{f}+1$ and $2\mathbf{f}+1$ threshold signatures, and are referred to as `RecoveryCC` and `StableCC`, respectively. Any non-prepared, non-faulty replica that receives the `RecoveryCC` message can restore its state and issues its own `CheckCommit` to the aggregator. Any replica that receives `StableCC` can conclude reaching the stable checkpoint state. To allow constructing both weak and strong certificates, each prepared replica signs its `CheckCommit` using both $\mathbf{f}+1$ and $2\mathbf{f}+1$ threshold schemes. A non-faulty replica that recovers through `RecoveryCC` message when issuing its `CheckCommit` would only sign using $2\mathbf{f}+1$ threshold scheme because the weak certificate has been created already by the aggregator.

What remains unanswered is how to cope with faulty aggregator, unreliable network, or an insufficient number of `CheckCommit` messages. Similar to the non-linear CheckCommit variant, the protocol relies on in-order processing implying that no replica will issue `CheckCommit` message unless all of its preceding rounds have been prepared since the last stable checkpoint. Given the cumulative implication of `CheckCommit`, the `RecoveryCC` and `StableCC` certificates for the round ρ_j aggregated by R_i is an implicit recovery for all rounds smaller and equal to j. Thus, the success of R_i compensates for the failure of earlier aggregators, whether due to faulty behavior or network unreliability as demonstrated in Figure 6.12.

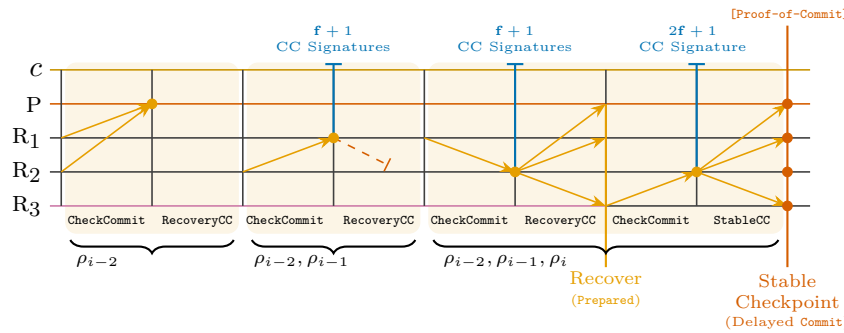


Figure 6.12: A schematic representation of linear CheckCommit protocol with failures such that each replica is designated as a `CheckCommit` aggregator in a round-robin fashion, in which R_i is assigned to the round ρ_j, where $i = j \bmod n$. The round-to-aggregator assignment is as follows: ρ_{i-2} to R_1, ρ_{i-1} to R_2, and ρ_i to R_3. P obtains sufficient number of `CheckCommit` messages but does not send `RecoveryCC` exhibiting faulty behavior. R_1 obtains sufficient number of `CheckCommit` messages supported by $\mathbf{f} + 1$ threshold signatures but its `RecoveryCC` are lost due to an unreliable network. R_2 obtains sufficient number of `CheckCommit` messages supported by $\mathbf{f} + 1$ threshold signatures and successfully broadcast `RecoveryCC` messages to re-affirms that the rounds $\rho_{i-2}, \rho_{i-1}, \rho_i$ were successfully prepared. Subsequently, R_3 sends `CheckCommit` message to R_2 enabling R_2 to construct and broadcast the commitment quorum via `StableCC` supported by $2\mathbf{f} + 1$ threshold signatures.

If the CheckCommit protocol continuously fails, then eventually, all replicas will stop accepting new proposals from the primary and simply fall back to the view change by issuing `ViewChange` messages. Therefore, the complexity of CheckCommit protocol is much simpler

than reaching consensus because CheckCommit simply commits in a decentralized way what was already proposed and prepared by the primary. If the CheckCommit is unable to progress, it simply blames the primary and replaces it. On the one hand, CheckCommit can commit and checkpoint proposals without relying on a primary, while, on the other hand, it will blame the primary when no progress is made.

To assess PoE objectively, the principle ideas in PoE that advances Pbft are two-fold: (1) illustrating that the commit phase can safely be removed from the critical path of consensus [62] and (2) introducing the notion of a rotating aggregator as the basis for designing checkpoint protocols in order to reach the commitment state in a completely decentralized manner with linear communication [74].

6.3.2 Optimistic Consensus: Dual-Path Commitments

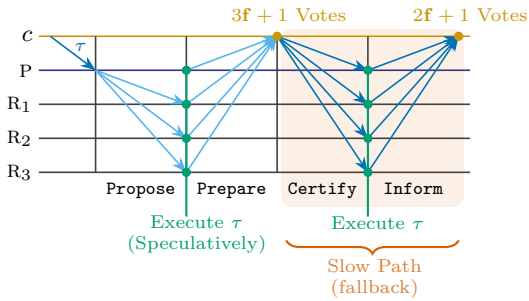
In addition to speculation, optimism can be introduced in consensus. This optimism is rooted in the following claim: *what if a failure is rare and what could happen if there are no failures at all?* If there is unanimous support for a proposal, referred to as the fast path of the protocol, could we potentially drop both prepare and commit phases? When the optimistic no-failure assumption is not held, then a slow fallback path can be instituted.

In Zyzzyva [94], albeit an unsafe protocol, both prepare and commit phases are removed from the fast path. If the client observes unanimous support, it concludes commitment through the fast path, and no further action is necessary. To accommodate the fast path, replicas speculatively execute the proposal as soon as the proposal is received from the primary and inform the client. However, suppose a client waits long enough (a parameter to be fine-tuned) but fails to receive unanimous support, then it falls back to the slow path by constructing a prepared quorum (assuming a quorum responded to the client after the prepare phase) and sending it to all replicas. If a quorum of replicas acknowledges the prepared quorum, then a non-faulty client can conclude commitment. Notably, on the fast path, we have unanimous support, while on the slow path, with fewer endorsements, a prepared certificate is formed. During recovery in Zyzzyva, a prepared certificate may naturally carry more weight and could overwrite possible unanimous support in an

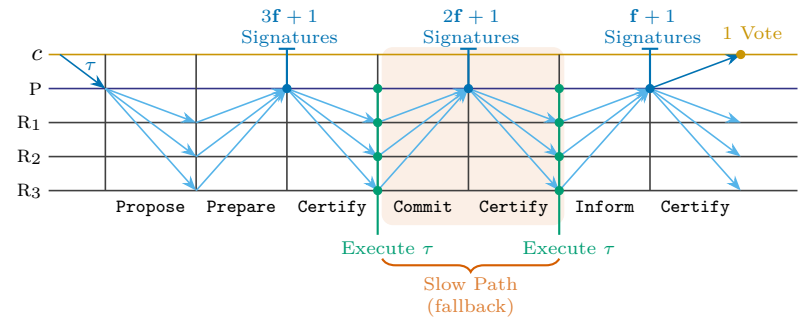
earlier round. It is possible that $\mathbf{f}$ faulty replicas may always retract their endorsements that led to unanimous support because their vote was never preserved in any exchanged certificate. This endorsement mismatch leads to the safety flaw of Zyzzyva as demonstrated in [4] and fixed in [5].

In a nutshell, consider a setting with four replicas consisting of an initial faulty primary F and three non-faulty replicas, $\mathrm{R}_1 \cdots \mathrm{R}_3$, and two clients c_a and c_b who propose transactions τ_a and τ_b, respectively. At round ρ_i, the primary equivocates by sending τ_a to R_1 and R_2 while sending τ_b to R_3. The client c_a receives $2\mathbf{f}+1$ endorsements from $\mathrm{F}, \mathrm{R}_1, \mathrm{R}_2$ and constructs a prepared certificate $\langle\!|\tau_a|\!\rangle_{c_a}$, which is received only by the faulty replica F before a view change is triggered due to network unreliability. During the view change, the primary P does not disclose the received prepared certificate for τ_a, and the new primary R_1 observes a recovery quorum consisting of $\mathrm{F}, \mathrm{R}_1, \mathrm{R}_3$ who claim τ_b, τ_a, τ_b were proposed at round ρ_i. The primary R_1 chooses and proposes τ_b that forces all non-faulty replicas who had observed τ_a previously at round ρ_i to roll back their speculative state. Subsequently, the client c_b receives unanimous support for τ_b with $3\mathbf{f}+1$ endorsements and considers τ_b as committed. Due to network unreliability, another view change is triggered, and the new leader R_2 observes the recovery quorum of $\mathrm{F}, \mathrm{R}_2, \mathrm{R}_3$, in which $\mathrm{R}_2, \mathrm{R}_3$ vote for τ_b while F revokes his vote for τ_b and now discloses the prepared certificate for τ_a that had been received in the first run of the protocol. The new leader R_2 oblivious to c_b's commitment favors the prepared certificate for $\langle\!|\tau_a|\!\rangle_{c_a}$ over $\mathbf{f}+1$ votes for τ_b; thus, failing to preserve the committed transaction τ_b, which in turn results in a critical safety violation, *i.e.*, commitment of both τ_a and τ_b at round ρ_i [4].

The safety flaw of Zyzzyva set the stage for the development of a new dual-path protocol, named, Sbft [54]. Its slow-path is essentially Pbft linearized using threshold signature, and its fast path is similar to Zyzzyva but it requires a replica such as the primary instead of the client to collect unanimous votes. Furthermore, the votes are aggregated using threshold signature in both fast and slow paths, which always results in forming a certificate to circumvent the mismatch that appeared in Zyzzyva. The flow of both Zyzzyva and Sbft protocols are demonstrated in Figure 6.13.



(a) Zyzzyva Protocol



(b) Sbft Protocol

Figure 6.13: (a) A schematic representation of the Zyzzyva protocol, in which the primary proposes the client's transaction τ to all replicas via a `Propose` message. Replicas commit to τ via one speculative phase of the all-to-one message directly to the client. Zyzzyva expects the client to collect $3\mathbf{f} + 1$ matching responses before concluding the commitment when a zero-failure optimistic route is plausible, *i.e.*, the fast path. The client falls back to the slow path in the presence of even a single failure and relies on a quorum of replicas to certify the speculative commitment through two linear phases of `Certify` and `Inform`. On the slow path, Zyzzyva expects the client to collect $2\mathbf{f} + 1$ matching responses before concluding commitment. (b) A schematic representation of the Sbft protocol, in which the primary proposes the client's transaction τ to all replicas via a `Propose` message. In the fast path, replicas speculatively prepare τ via two linear phases of `Prepare` and `Certify` while expecting unanimous support, $3\mathbf{f} + 1$ certify votes collected by the leader to construct the unanimous threshold signature (no client reliance). When the fast path fails, replicas may still commit to τ via two additional linear phases of `Commit` and `Certify` while expecting only support from the quorum of $2\mathbf{f} + 1$ replicas, and the leader constructs a weaker-quorum of threshold signature. Once τ is committed (whether through a slow or fast path), the execution is certified via another two linear phases of `Inform` and `Certify` but expecting only support from a quorum of $\mathbf{f} + 1$ replicas to sign, and the leader constructs even a weaker execution-quorum of threshold signature. The client is required to receive the execute certificate from a single replica.

6.3.3 Linearized Consensus: Rotating Leaders

By extending the linearized PBFT first introduced in the fast path of SBFT, we arrive at a new protocol called HOTSTUFF [157]. What sets apart HOTSTUFF from all previous protocols is that its commitment and recovery flows are combined and linearized. In POE and SBFT, only the commitment flow of PBFT was linearized.

To combine the commitment and recovery protocols, HOTSTUFF relied on rotating the leader after every commitment. This resulted in a flow consisting of four main stages of `Propose`, `Prepare`, `PreCommit`, and `Commit`[19] as demonstrated in Figure 6.14. At first glance, compared to PBFT, the combined flow introduces (1) a new `PreCommit` phase to lock the proposal explicitly and (2) leader rotation after every round via `NewView`. Having the replicas locked on proposals eliminates the need for all-to-all communication of what was previously prepared when changing the view. Only the latest prepared proposal is communicated with the next leader via `NewView`. This is an essential ingredient to linearize the message complexity when issuing `NewView`.

The new leader may optimistically initiate a new round of consensus, *i.e.*, *optimistic responsiveness*, by choosing the presumably latest prepared proposal as the basis without the need to first collect votes from a quorum. If the chosen prepared proposal extends an invalid or outdated locked proposal, then the leader's proposal will not receive sufficient endorsements to form a prepared quorum and will fail inevitably. A non-faulty replica would only endorse a proposal that extends its latest locked proposal. If a replica detects[20] that its own uncommitted locked proposal has already been overwritten and prepared by a newer proposal in a later view, then the replica releases its locked state and accepts the newer prepared proposal. Moreover, given that a new proposal may not directly extend the replica's latest prepared state, due to a multitude of reasons such as temporary network partition, then a non-faulty replica is forced to detect and fill the possible gap by consulting other

[19] In HOTSTUFF, these phases were referred to as `Prepare`, `PreCommit`, `Commit`, `Decide`, respectively, however, to ensure consistency in our unified model, we adopt the simpler naming convention.

[20] The detection is an external routine, which requires a replica to reconcile its states with others by gathering sufficient proofs such as committed certificates.

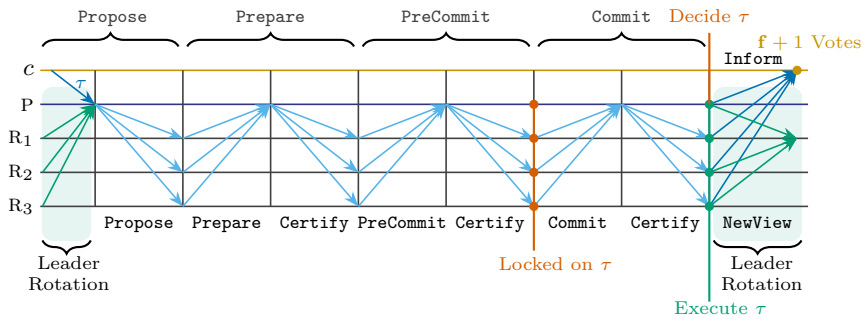


Figure 6.14: A schematic representation of the HotStuff protocol, in which the primary proposes client's transaction τ to all replicas in-order via a `Propose` message. The replicas prepare τ via two linear phases of `Prepare` and `Certify` while expecting support from the quorum of $2\mathbf{f}+1$ replicas, and the leader constructs the prepare threshold signature. Next, the replicas precommit τ via two linear phases of `PreCommit` and `Certify` while expecting support from the quorum of $2\mathbf{f}+1$ replicas, and the leader constructs the precommit threshold signature. Replicas lock on τ after receiving the precommitted certificate. A replica would only release its lock only if it receives a prepared certificate formed in higher rounds. Replicas finally commit to τ via yet another two linear phases of `Commit` and `Certify` while expecting only support from the quorum of $2\mathbf{f}+1$ replicas, and the leader constructs a commit threshold signature. Once τ is committed, the execution is carried out serially informing the client (who expects $\mathbf{f}+1$ endorsements), and more importantly, all committed (or timed out) replicas move to the next leader presenting their latest prepared certificate (their last locked state) via `NewView`.

non-faulty replicas. In an attempt to fill the gap without any prior knowledge, it is possible that the replica discovers that the proposed prepared extension was in fact invalid, which results in a loss of work. The provided latest prepared proposal could be invalid because the leader was faulty or due to failed optimism of choosing an incorrect proposal eagerly. Nevertheless, allowing a leader to choose the latest proposal without a quorum or certificate will give rise to a new plane of attacks by faulty leaders that were not feasible in Pbft.

Recall that recovery flow in Pbft consisted of three main steps: (1) detecting a leader failure, (2) agreeing on who the next leader is, a subtle yet critical synchronization step, and (3) agreeing on what was prepared in previous rounds. Most importantly, all these three steps—failure detection, leader replacement, prepared agreement—must be supported

by a recovery quorum. The quorum is constructed in PBFT through all-to-all quadratic communication. Now let's examine HOTSTUFF's attempt to linearize these steps. *First*, one may argue that HOTSTUFF is not concerned with failure detection as it rotates the leader after every round, so no recovery quorum is needed. *Second*, there is also no need to construct a recovery quorum to convince replicas of what was prepared in the previous round because no replica will accept any latest prepared proposal that does not extend its currently locked proposal. However, it is unclear how HOTSTUFF can ensure everyone agrees on who the next leader is or how a replica would know when it is time to begin acting as a leader in the presence of an unreliable network and faulty replicas.

In PBFT, the view is never advanced unless supported by a quorum, which implies that replicas endorsing the leader replacement will indefinitely stall and send a view change message unless a recovery quorum is formed. Thus, PBFT requires a phase with a quadratic communication in its view change to synchronize and agree on who the next leader is. To eliminate this quadratic step, HOTSTUFF abstracts out this vital synchronization step by encapsulating it in a black box referred to as a pacemaker. However, delegating a necessary synchronization step to a pacemaker does not eliminate its inherent quadratic complexity. One way to realize the pacemaker in practice is (1) to assume the existence of a global stabilization time after which the communication becomes reliable such that the protocol can operate correctly and (2) to weaken the communication model by assuming synchrony and reliable communication (*i.e.*, no message loss)—although a simple and valid theoretical model, its practical utility remains limited. The abstraction of the pacemaker gives rise to yet another important research question as to how to efficiently achieve a global stabilization time (*e.g.*, [26], [27])?

The leader rotation design in HOTSTUFF presents another interesting challenge as it strictly enforces a sequential consensus. Because every new proposal must extend the latest prepared proposal sequentially, which in turn disallows the out-of-order processing optimization. The out-of-order processing can partially be re-introduced by de-coupling the reliable dissemination of client requests from the critical path of

consensus as shown in [41], [88], [144]. However, in HotStuff, the cost of sequential consensus is further exacerbated due to adopting an eight-phase consensus flow after applying threshold signature because only a single proposal can be ordered at a time as it passes through all phases sequentially. In fact, the throughput would be limited by the network latency and determined by the length of the pipeline. Unavoidably, this may result in a long queue of unprocessed proposals and under-utilization of the network bandwidth.

To overcome the performance limitation of the sequential consensus design choice, the HotStuff protocol is further optimized to yield a chained variant as presented in Figure 6.15. This variant would shorten the sequential pipeline by reducing the number of explicitly required phases such that the round ρ_{i+1} would imply precommitment of the round ρ_i while the round ρ_{i+2} would mark the commitment of the round ρ_i. In other words, leaders are rotated after every phase of consensus as opposed to allowing a single leader to complete all phases of consensus. As such, every phase now constitutes as a new round for which the new leader must provide a new proposal (or no-op) extending the latest prepared proposal from the earlier round.[21] As a result, a single consensus decision requires the cooperation of at least three consecutive well-behaving leaders. Contrary to the design of Pbft, consensus liveness can no longer be guaranteed by relying on a single non-faulty leader [3], [26], [27]. Overall, the chained approach partially alleviates the loss of out-of-order processing by reducing the length of the sequential pipeline while prescribing to the optimistic assumption that the likelihood of observing consecutive non-faulty leaders is high despite the expectation that roughly $\frac{1}{3}$ of all replicas may behave maliciously. For example, in the original chained HotStuff, the liveness is completely lost in a setting consisting of four replicas where one of the replicas is malicious because the protocol requires four consecutive non-faulty leaders [3].

[21] Alternatively, the latest locked proposal could have been extended by the new leader instead of the latest prepared proposal. However, extending the latest lock will impede progress when adopting the chained HotStuff. It would translate into allowing only every other leader to provide a new proposal because it would take two non-faulty leaders and two rounds to lock a single proposal.

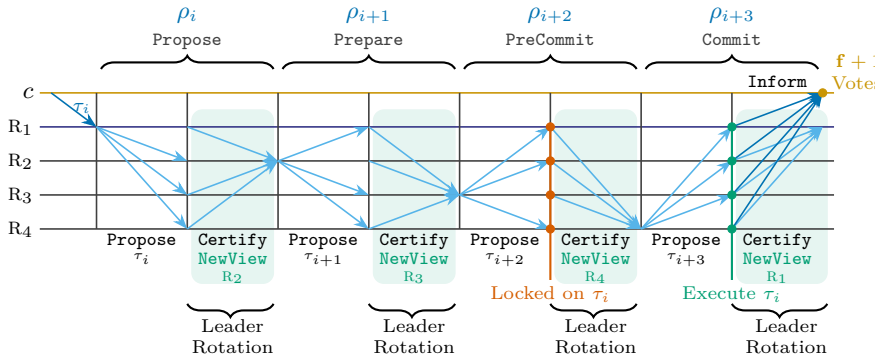


Figure 6.15: A schematic representation of the chained HotStuff protocol. In every round ρ_i, the leader is rotated. The new leader proposes the transaction τ_i to all replicas in-order via a `Propose` message. Each replica R_j certify τ_i via `Certify`, denoted by $\langle\!\langle \tau_i \rangle\!\rangle^{R_j}$. Although the τ_i is endorsed by all replicas in round ρ_i, the certificate of its prepared threshold signature can only be constructed by the next leader at round ρ_{i+1}, denoted by $\langle\!\langle \tau_i \rangle\!\rangle_{\rho_i}$. As a result, the transaction τ_i is prepared at round ρ_{i+1}, precommitted and locked at round ρ_{i+2}, and committed and executed at round ρ_{i+3}.

Now, let's examine HotStuff protocol fundamentally, framing a basic question as to whether there is a principle difference between Pbft and HotStuff.[22] At the surface level, HotStuff has an extra `Precommit` phase that explicitly introduces the notion of locking a proposal after observing a precommit quorum (cf. Figure 6.14). Furthermore, it invokes the `NewView` phase after every round of consensus to communicate what was prepared in the previous round. Invoking view change after every round can be applied to Pbft, as was studied in [29], [30], [32], [150], which can further be viewed as a matter of how the protocol is configured at runtime. So we focus on what does locking at precommit state entail.

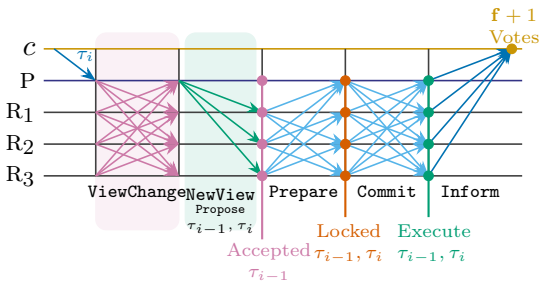
As explained earlier, Pbft strictly operates in lockstep, and replicas are implicitly locked at the prepared state, and the state can be unlocked in the view change phase if there is a recovery quorum that does not

[22] It is noteworthy that every consensus protocol covered here can be reduced to Pbft. Arguably since its inception [33], Pbft has been the pillar of every fault-tolerant consensus protocol that assumes a notion of identity to cast a vote.

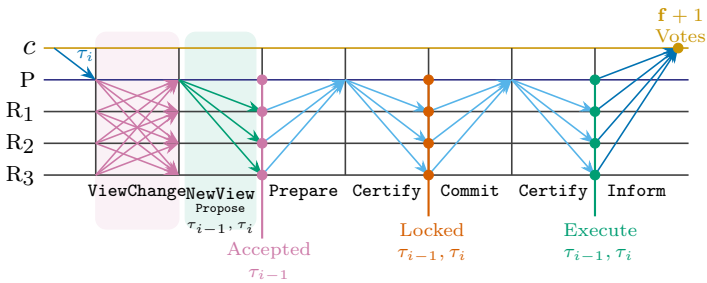
support the locked proposal. In contrast, replicas in HotStuff are locked in the precommit state, the stage after prepare, but they can also be unlocked during the view change if they are presented with a newer prepared proposal (supported by a quorum) that has overwritten their current locked state. So the question is why HotStuff introduces a new precommit phase that in essence delays acquiring the lock that was held by Pbft at the prepared state. We argue that difference is just a matter of semantics of our naming convention in how we label the different phases. We claim that the notable difference between Pbft and HotStuff is applying the threshold signature optimization to every single phase of the protocol. Recall, in Sbft and PoE, the threshold signature was only applied to the commitment flow of the protocol, but in HotStuff, the same optimization is also applied to the recovery flow.

We demonstrate our claim through a simple transformation of Pbft into HotStuff. We start from the basic Pbft protocol as shown in Figure 6.16a, where we first run a view change to recover what was prepared at the round ρ_{i-1} followed by running consensus for round ρ_i. To simplify the transformation, without loss of generality, we further assume that the new primary includes the proposal for the round ρ_i as part of its `NewView` message along with what it was prepared at ρ_{i-1}, this basic optimization was also adopted by HotStuff when constructing a prepared proposal. Next, in Figure 6.16b, we simply apply threshold signature to `Prepare` and `Commit` of Pbft. The final step is to apply the threshold signature to the `ViewChange` phase as shown in Figure 6.16c. Unlike the linearization of `Prepare` and `Commit` into two linear phases, the `ViewChange` phase transformation is subtle as it requires three linear phases. Because the new leader must first choose what was prepared in the previous round before certifying its selection through two linear phases of `Choose` and `Certify`.

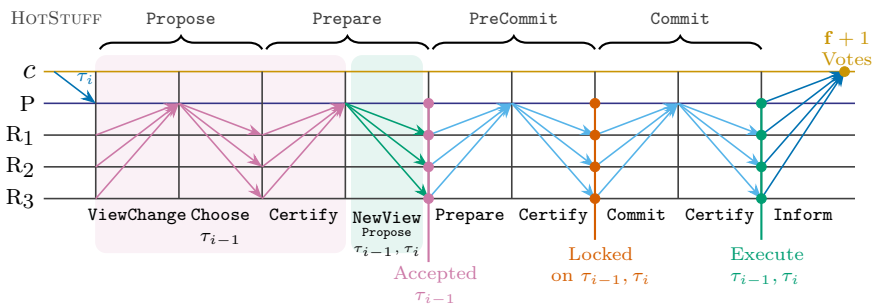
Intuitively, the reason why locking appears to be delayed in HotStuff is because what HotStuff calls `Precommit` is in fact Pbft's `Prepare` phase. Furthermore, HotStuff's `Prepare` is in fact the last step to linearize the view change phase to form a recovery quorum to endorse the newly proposed view. Thus, what HotStuff establishes in its `Prepare` is primarily to agree upon what was done in the previous view in addition to what is being proposed in the new round.



(a) PBFT Protocol with Recovery



(b) PBFT: Linearized Commitment Flow



(c) PBFT ↦ HotStuff: Linearized Recovery & Commitment Flows

Figure 6.16: (a) A schematic representation of the complete PBFT protocol as lockstep flow, in which τ_{i-1} was prepared at ρ_{i-1} while τ_i is being proposed in the round ρ_i. (b) A schematic representation of the complete PBFT protocol by applying threshold signature to linearize its commitment flow: `Prepare` and `Commit`. (c) A schematic representation of the complete reduction of HotStuff to the linearized PBFT protocol derived through a simple application of threshold signature to both commitment and recovery flows: `ViewChange`, `Prepare`, and `Commit`. Notably, the `ViewChange` phase discards the subtle time synchronization step of PBFT, which is abstracted out as a black box, referred to as a pacemaker, in HotStuff.

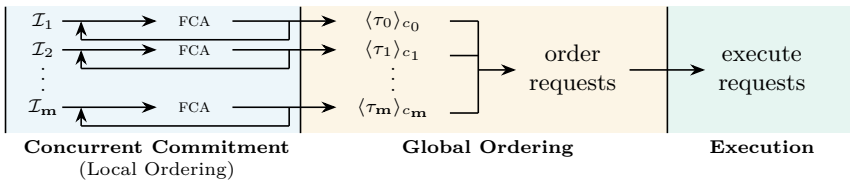
As rightly claimed in HotStuff, choosing what was previously prepared does satisfy *optimistic responsiveness* condition as the leader does not need to wait for a quorum before making its selection via `Choose`. However, to make any further progress, the leader relies on the endorsement of the quorum to certify its selection as part of the `Certify` phase. Hence, the notion of *optimistic responsiveness* no longer holds beyond the first phase of consensus.

6.3.4 Concurrent Consensus: Concurrent Commitments

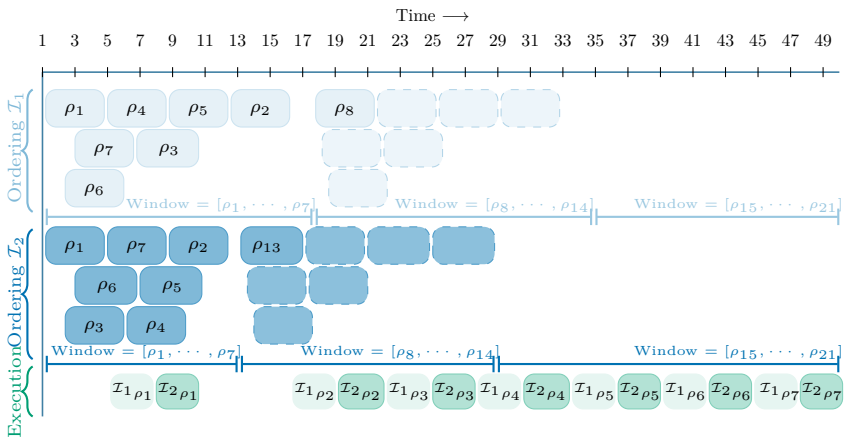
To move beyond improving the mechanics of running a single instance of a consensus protocol, we shift our attention to concurrent consensus, analogous to concurrency control protocols that are commonplace in database systems. The overarching aim is to allow running multiple instances of consensus concurrently with minimal coordination among them. In theory, this would multiply the system throughout by the total number of concurrent consensus, which would be similar to executing many concurrent transactions.

We choose Resilient Concurrent Consensus (Rcc) as a representative meta-protocol that may transform any single-primary consensus protocol into a multi-primary concurrent design [63], [65]. Rcc operates in rounds, where each round consists of three main phases of *local ordering*, *global ordering*, and *execution* as illustrated in Figure 6.17a. Rcc further assumes that a local single-primary consensus protocol could be separated into several independent subroutines such as a Fault-tolerant Commit Algorithm (referred to as fca) to order transactions; a checkpoint protocol to recover replicas in dark; and recovery or view change protocol to deal with misbehaving primary and an unreliable network.

Rcc exhibits a wait-free design by designating $\mathbf{m} > \mathbf{f}$ replicas as primaries, each running a local fca instance, in order to ensure there is at least a non-faulty primary that can always make progress. Furthermore, all replicas participate in all m instances, resembling running $\mathbf{m}$ independent and concurrent consensus in each round ρ. Since each instance operates independently, as soon as it completes its fca for round ρ_i, it will advance to locally order round ρ_{i+1} irrespective of the state



(a) Overview of RCC Protocol



(b) RCC with Out-of-Order Processing

Figure 6.17: (a) A schematic representation of the RCC meta-protocol that runs **m** concurrent instances of FAULT-TOLERANT COMMIT ALGORITHM (FCA) in a single round. Each instance independently and continuously operates. There are **m** replicas that act as primary running FCA to locally order its local transaction $\langle\tau_i\rangle_{c_i}$ received from its local client c_i. Each replica further participates as a backup in **m** instances. A round is considered to be completed once all m instances complete their local ordering, deciding on either a transaction τ_i or a no-op upon recovery. Given m decisions, they are ordered deterministically and executed serially by all replicas. (b) A schematic representation of supporting out-of-order processing within each instance of FCA. The primary of each instance $\mathcal{I}_i$ may initiate FCA on the round ρ_j before waiting for any of the previous rounds to be completed first. Although consensus can be carried out-of-order, the execution must be carried strictly in order across all instances such that the execution of ρ_i is only possible when all the preceding rounds for both instances have been executed. Each blue box represents a consensus step for the round ρ_i while each green box represents the execution step of the proposal decided in ρ_i. The lighter blue/green shades are associated with $\mathcal{I}_1$ while the darker shades are associated with $\mathcal{I}_2$. It is assumed that $\mathcal{I}_1$ is ordered before $\mathcal{I}_2$. The low-high watermark window size is 7.

of other concurrent consensus instances. Each instance may even enable out-of-order processing as demonstrated in Figure 6.17b. Although the local ordering is carried out independently and continuously, in each round, RCC must synchronize the global ordering to ensure consistent execution and state replication across all replicas. Therefore, within each round, once all local FCAs are completed, then a deterministic and consistent ordering is adopted by all replicas. Once the global ordering is established, each replica independently and serially executes all ordered transactions within each round similar to PBFT. Unlike the ordering, the execution of round ρ_i must always precede the round ρ_{i+1}.

In any given round, if the local FCA fails to reach commitment, then the recovery protocol is triggered. The recovery itself would be a new independent consensus instance that aims to recover the state of the failed instance. However, unlike traditional view change protocol, it does not attempt to replace the primary. Instead, it temporarily stops the primary with an exponential back-off. To avoid global coordination during recovery, RCC adopts a fixed assignment of primary to each instance. Thus, primaries are only stopped and never replaced.

In general, any multi-primary consensus design must cope with collusion among primaries, an attack that was not present in PBFT. In RCC, primary collusion may go undetected, resulting in a loss of liveness. For example, given two replicas R_j and R_k, the colluding primaries for instances j and k may prevent R_j to progress on instance k while preventing R_k to make progress on instance j. Although each instance may appear to be live locally, neither R_j and R_k can progress as far as the global ordering and execution are concerned; thus, creating a deadlock stalling both instances from execution.

A checkpoint protocol allows the recovery of any replicas kept in dark by the colluding primaries. By periodically running the checkpoint protocol within each instance independently, RCC ensures that no replicas are kept indefinitely in dark, and eventually, the execution can ensue.

6.4 Consensus Topology

The next design point in our repertoire is to scale consensus by examining the topology of consensus, namely, partitioning replicas into a set of disjoint committees or clusters, where each cluster may maintain a full or partial replication of the log.

6.4.1 Cross-Cluster Communication Primitive

Before inspecting the consensus partitioning idea, we identify a central problem as to how to efficiently and reliably communicate between any two clusters, where a cluster may have up to $\mathbf{f}$ faulty replicas. Of course, the communication step can always be solved trivially as a consensus step using PBFT or its variant. Yet the question remains as to whether a reliable cluster communication has a lower computational complexity cost than consensus.

The problem of sending a message reliably from one cluster to another is formalized as CLUSTER SENDING PRIMITIVE (CSP) in [76], [79]. The basic idea of CSP is to ensure that at least one non-faulty replica from the source cluster sends the message to another non-faulty replica in the destination cluster. This observation serves as the basis to establish the lower-bound of linear message complexity. Furthermore, an optimal bijective cluster-sending algorithm is developed, which guarantees at least a pair of non-faulty replicas will successfully communicate by requiring at most $2\mathbf{f} + 1$ distinct pairwise communications when the network is reliable, as shown in Figure 6.18.

The problem of CSP is further analyzed probabilistically such that instead of invoking $2\mathbf{f} + 1$ pairwise communications in parallel, one can randomly choose a pairwise communication at a time until the message is communicated successfully [78]. The probabilistic variant is shown to have constant inter-cluster message complexity in expectation reduced from the original linear complexity.

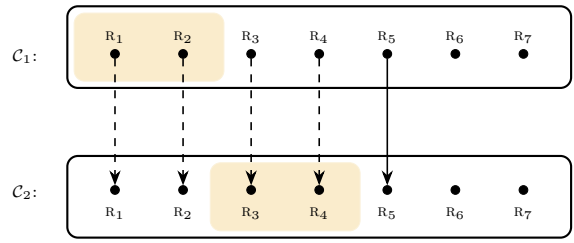


Figure 6.18: A schematic representation of Cluster Sending Primitive (Csp) to send a message reliably from $\mathcal{C}_1$ to $\mathcal{C}_2$, in which each $\mathcal{C}_i$ is of size $3\mathbf{f}+1$ with at most $\mathbf{f}=2$ faulty replicas. The bijective sending between two clusters relies on pairwise communication, where each replica in the source cluster communicates with a distinct replica in the destination cluster. The faulty replicas are marked as red, and their communication is highlighted using dotted lines, while solid lines mark communication between non-faulty replicas. In the worst case, $2\mathbf{f}+1$ pairwise communications are needed (assuming a reliable network) to ensure a message is exchanged between at least one pair of non-faulty replicas, establishing the lower-bound as linear message complexity.

6.4.2 Global Consensus: Full Replication

One way to restructure consensus is to form clusters of replicas and to partition client workload among these clusters while retaining a fully replicated model. In theory, all clusters could operate in parallel to independently order their clients' proposals locally, then replicate the local order globally.

Csp may serve as the basis to design a global consensus protocol over clusters of replicas that are geographically distributed across data centers. We assume that each cluster is maintained independently yet holds a full copy of data. The problem of global consensus can be reduced to maintaining a fault-tolerant globally consistent replication managed through local consensus. In particular, we examine GeoBFT as a meta-protocol in a fully replicated setting such the client workload is partitioned based on the network topology and clients' proximity to each cluster [66].

Clustered Setting: We model a *topological-aware system* as a partitioning of $\mathfrak{R}$ into a set of clusters $\mathfrak{C}=\{\mathcal{C}_1,\ldots,\mathcal{C}_\mathbf{z}\}$, in which each cluster $\mathcal{C}_i$, $1\leq i\leq \mathbf{z}$, is a set of $|\mathcal{C}_i|=\mathbf{n}$ replicas of which at most $\mathbf{f}$ are *faulty* and can behave *maliciously* and possibly coordinated. We write $\mathbf{n}_{\mathcal{C}_i}=|\mathcal{C}_i|$, $\mathbf{f}_{\mathcal{C}_i}=|\mathcal{F}(\mathcal{C}_i)|$, and $\mathbf{g}_{\mathcal{C}_i}=|\mathcal{G}(\mathcal{C}_i)|$ to denote the number of

replicas, faulty replicas, and good replicas in each cluster, respectively. We assume that in each cluster $\mathbf{n}_\mathcal{C} > 3\mathbf{f}_\mathcal{C}$. Without loss of generality, we further assume that a set of *clients* are partitioned dis-jointly across the set of clusters $\mathfrak{C}$.

GeoBFT operates in rounds, and within each round ρ, clusters accept clients' transactions and order them locally using any consensus protocol. Next, the local ordering is exchanged among clusters using Csp. Once all local decisions are exchanged, a deterministic global ordering is computed, similar to Rcc.[23] Each cluster will execute all transactions serially per the global order.

In each round, every cluster expects to receive a local decision from all other clusters or at least a no-op heartbeat in the absence of any clients' transaction. Thus, the cluster-sending step can further be optimized optimistically by assuming the primary of each cluster is non-faulty and faithfully sends the local decision to at least $\mathbf{f}+1$ replicas in all remote clusters. This further reduces the number of global messages further from $2\mathbf{f}+1$ to $\mathbf{f}+1$. If the primary fails to transmit the messages for any reason, *e.g.*, faulty intent or unreliable communication, then eventually, the remote cluster will timeout and invoke a remote view change to replace the misbehaving remote primary to restore the liveness. While a cluster is waiting for remote messages, it may continue its local ordering for subsequent rounds as out-of-ordering can be utilized. However, the execution stalls as it must be done in order. The overall flow of GeoBFT is presented in Figure 6.19.

6.4.3 Sharded Consensus: Partial Replication

To scale parallelism one step further, the replicated state can be partitioned into shards. This gives rise to a partially replicated design, in which each cluster (or a shard) holds only a slice of the log. Thus, the execution of a single transaction may span multiple shards, which requires a coordination mechanism among shards. First, we formally extend our model to the sharded setting.

[23] Noteworthy, there is no need to exchange local consensus decisions in Rcc because the same set of replicas runs all instances. In other words, unlike GeoBFT that spans multiple clusters, Rcc runs over a single cluster.

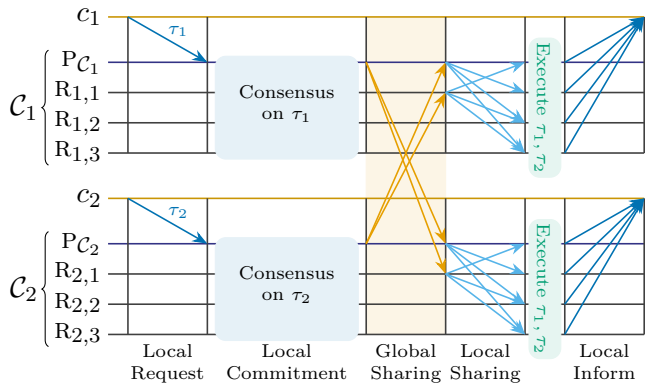


Figure 6.19: A schematic representation of the GeoBFT protocol over two clusters $\mathcal{C}_i$, $i \in \{1, 2\}$. Each cluster $\mathcal{C}_i$ independently run local consensus (*e.g.*, Pbft) to order the transaction τ_i issued by its client c_i. Once the local commitment is completed, the primary $P_{\mathcal{C}_i}$ in each cluster $\mathcal{C}_i$ constructs a committed certificate for τ_i and share it via a linear inter-cluster communication primitive, *i.e.*, Csp. In each round, after the completion of the global sharing phase, all clusters execute all ordered transaction τ_i serially in the same order and inform their local clients.

Sharded Setting: We model *sharded system* as a partitioning of $\mathfrak{R}$ into a set of $\mathbf{z}$ shards $\mathfrak{S} = \{\mathcal{S}^1, \ldots, \mathcal{S}^{\mathbf{z}}\}$. Let $\mathcal{S}^i \in \mathfrak{S}$ be a shard. We write $\mathbf{n}_{\mathcal{S}^i} = |\mathcal{S}^i|$ to denote the number of replicas in each $\mathcal{S}^i$ and $\mathbf{f}_{\mathcal{S}^i} = |\mathcal{F}(\mathcal{S}^i)|$ to denote the faulty replicas in each $\mathcal{S}^i$. We assume $\mathbf{n}_{\mathcal{S}} > 3\mathbf{f}_{\mathcal{S}}$. Let τ be a transaction. We write $shards(\tau) \subseteq \mathfrak{S}$ to denote the shards that are relevant to τ that hold the data that τ needs to read or write. We write $|shards(\tau)|$ to denote the number of shards involved in τ.

The central problem in sharded setting can be reduced to studying cross-shard coordination patterns inspired by 2pc. Conceptually, the coordination in 2pc is nothing but a method to aggregate vote results in an election. The cross-shard coordination pattern is classified as *linear*, *centralized*, or *decentralized* through ByShard formalism [77]. These intuitive methods to collect votes are referred to as *orchestration patterns* in ByShard and described as follows.

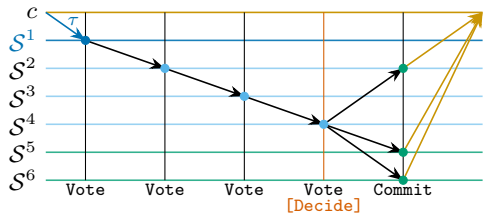
- *Linear*: To construct a chain of shards, where each shard forwards its vote to the next shard in accordance to the shard priority order until the last shard in the chain can tally the vote results (*e.g.*, RingBFT [135])

- *Centralized*: To designate a single shard to collect all votes from all shards (*e.g.*, AHL [42]).
- *Decentralized*: To instruct all shards to broadcast their votes to all shards so that each shard can independently tally the votes (*e.g.*, PESSIMISTIC CERBERUS [75])

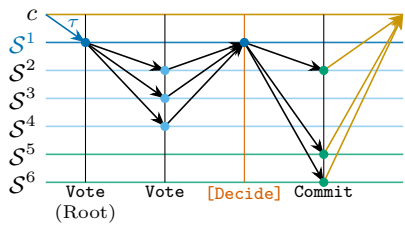
The orchestration patterns provide the means to tally and collect votes in order to reach a decision and communicate it with all involved shards as presented in Figure 6.20. What distinguishes these patterns is the trade-off between the number of sequential vote-steps versus the number of cluster-sending steps (cf. CSP). Suppose the transaction τ is involved in $|shards(\tau)| = \mathbf{k}$ shards. In linear orchestration, the number of sequential steps to commit a decision is proportional to the number of shards involved, which is $\mathbf{k}$ sequential vote steps, and, similarly, the number of cluster-sending steps needed to reach a decision is also $\mathbf{k}$. In centralized orchestration, the number of sequential steps to commit a decision is constant and fixed at 4, while the number of parallel cluster-sending steps needed to reach a decision is $2\mathbf{k}$. In decentralized orchestration, the number of sequential steps to commit a decision is constant and fixed at 3, while the number of parallel cluster-sending steps needed to reach a decision is $\mathbf{k}^2$. Intuitively, we conclude that the linear approach is optimized for throughput at one end of the spectrum by reducing the number of costly cluster-sending steps. In contrast, at the other end of the spectrum, the decentralized approach is optimized for latency at the cost of a quadratic number of cluster-sending steps.

In BYSHARD, each shard may further contribute to the processing of transaction at a different capacity, modeled as *vote-step* and *action-step*. A vote-step is intended to endorse whether a transaction should be executed or not, for example, checking integrity constraints and acquiring read or write locks. An action-step is tied to commit or abort decision that is intended to make writes visible and release locks. In BYSHARD, the complexity of these patterns is further analyzed in conjunction with a wide range of consistency and isolation semantics, *e.g.*, dirty reads, committed reads, and serializability.

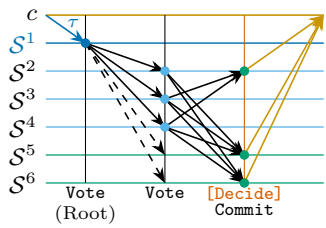
As a representative in the *linear orchestration* space, we examine RINGBFT, which is a meta-protocol for sharded consensus that requires



(a) Linear Orchestration



(b) Centralized Orchestration



(c) Distributed Orchestration

Figure 6.20: A schematic representation of the BYSHARD orchestration patterns over six shards $\mathfrak{S} = \{\mathcal{S}^1, \cdots, \mathcal{S}^6\}$ in order to execute a cross-shard transaction τ with $shards(\tau) = \{\mathcal{S}^1, \ldots, \mathcal{S}^6\}$. We assume that $\mathcal{S}^1$, $\mathcal{S}^2$, $\mathcal{S}^3$, and $\mathcal{S}^4$ have vote-steps while $\mathcal{S}^2$, $\mathcal{S}^5$, and $\mathcal{S}^6$ have commit steps. Each dot represents a single local consensus step and each arrow represents a cluster-sending step. It may be sufficient to have a single shard informing the client. (a) A schematic representation of linear orchestration, in which the orchestration begins at $\mathcal{S}^1$ and the final commit decision is determined at $\mathcal{S}^4$ after three linear vote steps of $\mathcal{S}^1$, $\mathcal{S}^2$, and $\mathcal{S}^3$. (b) A schematic representation of centralized orchestration, in which the orchestration is coordinated and the final commit decision is determined by $\mathcal{S}^1$ after three parallel vote steps by $\mathcal{S}^2$, $\mathcal{S}^3$, and $\mathcal{S}^4$. (c) A schematic representation of distributed orchestration, in which the orchestration is initiated at $\mathcal{S}^1$ followed by three parallel vote steps by $\mathcal{S}^2$, $\mathcal{S}^3$, and $\mathcal{S}^4$. The final commit decision is arrived independently by all shards in parallel. In the distributed orchestration, each dashed black arrow represents a cluster-sending step in which the root shard sends its vote to all the involved shards without vote-steps.

all shards to adhere to a predetermined ring order [135]. It follows three basic principles of *process*, *forward*, and *re-transmit* while ensuring the communication between shards is linear. Conceptually, RingBFT guarantees consensus for each cross-shard transaction in at most two rotations around the ring. Each shard may participate in multiple circular flows simultaneously. This implies that each shard performs consensus (*i.e.*, *process*) on transactions independently before *forwarding* it to the next shard. This flow continues until each shard is aware of the fate of the transaction. The flow of RingBFT is depicted in Figure 6.21.

The real challenge of processing cross-shard transactions is managing conflicts and preventing distributed deadlocks. To this end, RingBFT assumes that the read/write sets of each transaction are known as *a priori*, an approach that is widely adopted in deterministic databases [137]. RingBFT requires all cross-shard transactions to traverse each involved shard in the ring order. The transaction is forwarded to the next shard only if the read/write set can be locked at each shard in order to prevent deadlocks. RingBFT forwarding relies on the linear cluster-sending steps (cf. Csp) that exhibits a neighbor-to-neighbor communication but unlike ByShard formulation does not assume reliable inter-cluster communication.

To cope with the unreliable inter-cluster communication setting, the recovery protocol of RingBFT is centered around three timers: *local timer*, *transmit timer*, and *remote timer*. A *local timer* is necessary to ensure the liveness of local commitment inside each shard, which can be delegated to local view change and leader replacement routines. However, to ensure cross-shard progress by successfully forwarding transactions to the next shard, even in the presence of an adversarial network, then additional timing mechanisms are required.

Unique to RingBFT, once a shard forwards a transaction using the cluster-sending steps, each sending replica starts its *transmit timer*. If the transmit timer expires before any commit decision is received, the sender will indefinitely re-transmit the message and exponentially increase its transmit timeout values. Therefore, the transmit timer is utilized to detect and overcome challenges arising from the unreliable network from the sender's perspective. However, if the forwarding step is partially successful, as soon as the receiving replicas can partially

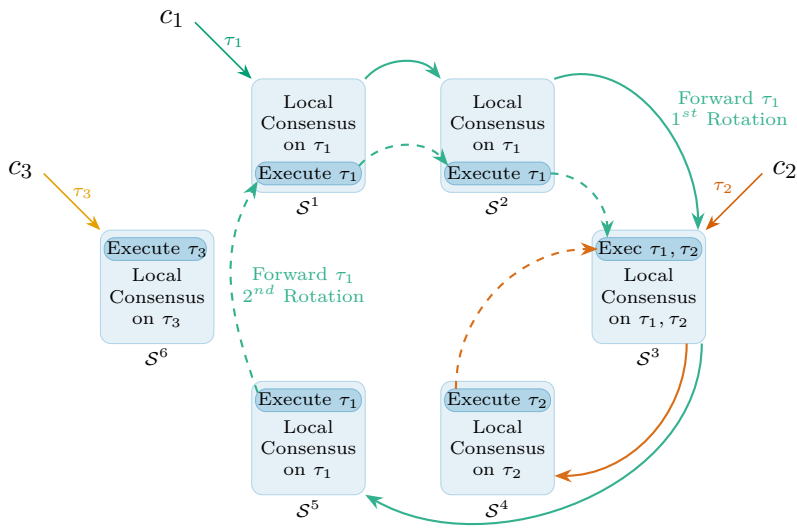


Figure 6.21: A schematic representation of the RingBFT protocol over six shards $\mathfrak{S} = \{\mathcal{S}^1, \cdots, \mathcal{S}^6\}$. Each shard $\mathcal{S}^i$ is assigned a predetermined priority ring order i such that $\mathcal{S}^i$ has a higher priority than $\mathcal{S}^j$ when $i < j$, which results in the following ring order $\mathcal{S}^1 \rightarrow \mathcal{S}^2 \rightarrow \mathcal{S}^3 \rightarrow \mathcal{S}^4 \rightarrow \mathcal{S}^5 \rightarrow \mathcal{S}^6$. In the first rotation of RingBFT, each shard $\mathcal{S}^i$ independently runs a local consensus to order its transactions followed by locking its read and write sets. Once the local commitment is completed, the transaction is forwarded via a linear inter-cluster communication primitive to the next involved shard in the ring order. The last involved shard will determine the commit or abort decision, inform the client, and initiate the second rotation to propagate the final decision, installing writes, and releasing locks in all involved shards. The example illustrates how RingBFT concurrently orders two cross-shard transactions $shards(\tau_1) = \{\mathcal{S}^1, \mathcal{S}^2, \mathcal{S}^3, \mathcal{S}^5\}$ and $shards(\tau_2) = \{\mathcal{S}^3, \mathcal{S}^4\}$ and a single-shard transaction $shards(\tau_3) = \{\mathcal{S}^6\}$, proposed by the clients c_1, c_2, c_3, respectively. Notably, if the concurrent transactions τ_1 and τ_2 are conflicting at $\mathcal{S}^3$, then whichever transaction that does not arrive first is held back until the execution of the first transaction is completed and locks are released.

observe the state of the previous shard, they start their *remote timers*. A partially forwarded state of a transaction implies that a transaction was ordered in the previous shard, but it is unclear whether or not all read and write locks could have been acquired. If the remote timer expires before the full state of the previous shard is forwarded, the receiving replica will indefinitely trigger a remote view change to replace the primary of sending shards.

Other notable sharding protocols are Ahl [42], SharPer [10], and Cerberus [75]. Ahl, as a representative in the *centralized orchestration* scheme, introduced the notion of a reference committee, itself a set of replicas, as a centralized coordination entity to essentially run a fault-tolerant two-phase commit protocol. Pessimistic Cerberus adopts a *decentralized orchestration* design, in which each shard independently reaches consensus on a fragment of the transaction for which it is responsible and communicates the local decision to all involved shards using a cluster-sending step (cf. Csp). Thus, each shard can unilaterally decide the fate of the transaction once it receives decisions from all involved shards. Consequently, a global view change is triggered to deal with misbehaving clusters and network unreliability.

Alternatively, SharPer adopts a semi-decentralized model such that one of the involved shards is designated as the initiator. Through the initiator, all involved shards are logically seen as a single super-shard on which a Pbft-like protocol is ran. Thus, SharPer relies on all-to-all quadratic communication among all replicas of all the involved shards. To cope with concurrent conflicting transactions, a coarse-grained locking at the shard level may be employed. Optimistic Cerberus also relies on the notion of a logical super shard that encompasses all involved shards to run a local consensus for a consistent cross-shard ordering. Optimistic Cerberus does not promote any shard as initiator and avoids any coarse-grained locking by leveraging the unspent transaction model in [124], but its optimism may result in higher abort rates due to contention among conflicting concurrent transactions. Both SharPer and Optimistic Cerberus deviate from the orchestration patterns in ByShard due to the formation of logical super-shard and not leveraging the clustering-sending primitive as the only mode of linear communication among shards. In fact, adopting a notion of single logical super-shard in a sense eliminates the need for having 2pc's style vote-collection from multiple involved shards.

6.5 Permissionless Consensus: Membership Model

In the classical formulation of consensus in the permissioned setting, we assume the existence of a verifiable identity, *i.e.*, *closed membership*,

and we adopt PBFT-like protocols. The existence of identity is necessary because these protocols rely on counting votes to construct a quorum, forming the majority. Without a verifiable identity, the entire concept of election and voting is nullified because many fake identities can be generated to dilute the election outcome, which is also known as the Sybil attacks. But in the public permissionless setting, we rely on PROOF-OF-WORK (POW) protocols [124] and assume the absence of a central authority to issue verifiable identity, *i.e.*, *open membership*.

In PBFT, the barrier to participate or vote is the existence of a verifiable identity; however, in POW, the barrier to voting is the proof of work instead of the proof of identity. Before any participant can vote in any round of POW consensus, they must solve a compute-intensive puzzle, which makes the creation of fake identities to dilute the election prohibitively expensive. As a result, after each round of consensus, PBFT will produce a verifiable certificate signed by the majority, while in POW, the certificate is the solution to a puzzle, the proof of work. Therefore, in the absence of identity, solving a difficult puzzle may be substituted as a viable alternative.

Unfortunately, the energy and resources loss from solving arbitrary random puzzles is unsustainable, further destroying our already fragile environment and climate. This has led to the development of rich array of PROOF-OF-X protocols to reduce the energy cost such as PROOF-OF-STAKE [91], PROOF-OF-ACTIVITY [19], CHAINS-OF-ACTIVITY [18], [90], DELEGATED PROOF-OF-ACTIVITY [25], BONDED PROOF-OF-ACTIVITY [32], [111], PROOF-OF-SPACE [14], [47], [130], and POWER-OF-COLLABORATION [39].

Furthermore, there is an abundance of clever optimization to scale throughput such as sharding techniques that are adapted to the permissionless setting [92], [113], [161]. An important abstraction adapted from the sharded setting is atomic cross-chain transactions with the key difference that we expect each chain (*i.e.*, shard) is maintained by independent parties or consortia who may not necessarily trust each other. Nevertheless, the overarching agreement problem can still be realized by relying on 2PC paradigm or its BYSHARD generalization. So what arguably distinguishes cross-chain from cross-shard is the certificate that is produced as part of the local ordering within each chain.

The BYSHARD formulation is sufficiently expressive to represent a variety of cross-chain protocols because the key distinction that separates cross-chain from cross-sharing is how PROOF-OF-X certificate is utilized instead of a notarized certificate signed by a quorum of replicas. The second differentiation is how the certificate is exchanged, which can be facilitated through either a push or pull mechanism. The push can be realized through CLUSTER SENDING PRIMITIVE while the pull can be viewed as retrieving the certificate directly from a common or shared witness chain (*i.e.*, ledger).

For example, the atomic cross-chain protocol presented in [158] resembles a centralized orchestration design of BYSHARD in which the central coordinator is modeled as a witness chain. Its basic design dictates logging each transaction's initiation and final outcome explicitly, *i.e.*, the commit or abort decision. The final decision can be provided by any shard to the witness chain, acting as a central coordinator, as long as verifiable and settled PROOF-OF-X-certificate from all shards supporting the decision is written to the witness. In contrast, to explicitly logging the final outcome on the witness chain, an alternative CERTIFIED BLOCKCHAIN (CBC) protocol is developed in [80] such that the initiation of the transaction and the individual votes of each shared are logged onto the witness chain, again serving as central coordination medium to tally votes. All valid shard-votes must be supported by a PROOF-OF-X certificate. The final decision can easily be determined if all involved shards voted to commit and abort otherwise.

7

Conclusion

In this monograph, we provided an overview of the problem of consensus and its use in data management systems. We covered the basic principles and reference protocols for consensus solutions that are widely used. Then, we described their use in two of the most important problem in distributed data management: distributed atomic commit and data replication. Finally, we provided an overview of consensus in fault tolerant environments where malicious and arbitrary faults may occur. This is an area of increased interest due to emerging blockchain applications. We described how consensus can be solved in such a setting and overviewed various challenges and optimizations faced by these protocols.

We conclude this monograph with a brief discussion of anticipated future directions and open problems in the area of applying consensus to data management systems.

Consensus in Serverless Environments

An important recent development in cloud computing is the emergence of the serverless compute model [84]. This model aims to simplify the utilization of cloud resources. Instead of having to manage a virtual machine, a programmer deploys their application functions and opera-

tions. The serverless framework, then, utilizes the deployed functions to answer incoming requests for the application. In this *Function-as-a-Service (FaaS)* model, the programmer is only charged when their functions/applications are called and the provisioning of resources to answer such requests is performed transparently.

To deliver the attractive serverless features to programmers, certain limitations need to be enforced on the nature of deployed functions and serverless workers that run these functions. One such limitation is that the function is stateless, meaning that it needs to coordinate with other services to manage any persistent state. This creates a challenge in coordinating the processing and operation of different serverless workers that are working on the same application and may perform conflicting operation.

Applying traditional consensus algorithms in serverless environments is not straightforward due to these limitations [73]. Therefore, there is a need for work on consensus algorithms that overcome these limitations and provide solutions that can operate in serverless environments. This entails enabling consensus protocols to react seamlessly to changes in the system model and configuration of the system. Such protocols can be integrated to enable consensus on a dynamic set of serverless workers that are continuously changing for each application. Towards that direction, consensus protocols that emphasize efficient reconfiguration [153] and dynamic quorums [8], [82], [126] can be used as initial designs to build on. Also, integrating relaxed consistency abstractions with consensus solutions can offer a way to adapt to the highly dynamic nature of serverless workers. This direction can build on Works such as MDCC [95] that explore building data abstractions with relaxed consistency over consensus algorithms.

Serverless environments will likely be part of larger data ecosystems that contain traditional cloud services as well as external client environments. This introduces interesting research challenges in terms of developing consensus protocols that enable coordination across these different types of infrastructures. ServerlessBFT [67] proposes a consensus protocol to manage agreement of systems that utilize serverless technology while spanning edge and cloud environments. Such a protocol needs to balance between the trade-offs of different environments in terms of resources, trust, and functionality.

Consensus for Decentralized Smart Contract Environments

The emergence of decentralized application (DApp) development raises interesting opportunities and challenges on how to coordinate state and interactions between the users of such applications. DApps are applications that are deployed on blockchain infrastructures where users can interact with the application by issuing requests to the corresponding smart contracts [12]. This enables users to interact with the application and its state that is preserved in the blockchain smart contract. This abstraction can be utilized to coordinate between users. For example, a smart contract can make decisions that pertain to a distributed system or group of clients. This presents an opportunity to provide a single source of coordination for distributed or decentralized systems/users.

However, the success of this model faces various challenges that need to be addressed. One of these challenges is the cost and overhead involved in interacting with DApps and smart contracts—especially in public permissionless blockchains. Another set of challenges is due to the security implications of coordinating through a transparent infrastructure. Private data, for example, needs to be encrypted or hidden, which may impact the complexity of the consensus process. Also, users have access to the pool of transactions that are admitted to miners, which presents the risk of malicious actors taking advantage of this knowledge. For example, a malicious actor may observe the votes/requests of other users and construct a vote/request accordingly. This constructed request can even be pushed to be ordered before the previously observed requests by paying higher fees to miners. This pattern can lead to various attacks. Overcoming these challenges is an active area of research that spans work on different layers of the blockchain and DApp development stack.

Consensus for Edge-Cloud Environments

Emerging IoT and edge applications motivate the utilization of edge resources for faster response times, better privacy and data regulation, and to reduce the bandwidth utilization to centralized data centers. This introduces the need for data management systems that span both edge and cloud resources. Such systems pose interesting research challenges

due to the asymmetry of resources in the edge and the cloud. Designing distributed protocols and consensus mechanisms that account for this asymmetry has the potential of utilizing edge resources efficiently. Work in this area includes proposing consensus protocols that aim to provide better support for hierarchical and locality aware design [126]. Also, it includes the support for coordination mechanisms that distinguish between processing in the edge and processing in the cloud [51], [119], [125].

References

[1] M. Abebe, B. Glasbergen, and K. Daudjee, "Dynamast: Adaptive dynamic mastering for replicated systems," in *36th IEEE International Conference on Data Engineering (ICDE)*, 1381–1392. (2020), 2020.

[2] M. Abebe, B. Glasbergen, and K. Daudjee, "Morphosys: Automatic physical design metamorphosis for distributed database systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 13, 2020, 3573–3587. (2020).

[3] I. Abraham, N. Crooks, N. Giridharan, H. Howard, and F. Suri-Payer, "Brief announcement: It's not easy to relax: Liveness in chained BFT protocols," in *36th International Symposium on Distributed Computing (DISC)*, vol. 246, 39:1–39:3. (2022), 2022.

[4] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J. Martin, "Revisiting fast practical byzantine fault tolerance," *CoRR*, vol. abs/1712.01367. (2017), 2017. arXiv: 1712.01367.

[5] I. Abraham, G. Gueta, D. Malkhi, and J. Martin, "Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma," *CoRR*, vol. abs/1801.10022. (2018), 2018. arXiv: 1801.10022.

[6] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology (MIT), Deptartment of Electrical Engineering and Computer Science (EECS). (1999), 1999.

[7] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases," in *European Conference on Parallel Processing (Euro-Par)*, 496–503. (1997), 1997.

[8] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar, "Wpaxos: Wide area network flexible consensus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, 2019, 211–223. (2019).

[9] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, 562–570. (1976), 1976.

[10] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "SharPer: Sharding permissioned blockchains over network clusters," in *ACM International Conference on Management of Data (SIGMOD)*, 76–88. (2021), 2021.

[11] M. J. Amiri, C. Wu, D. Agrawal, A. E. Abbadi, B. T. Loo, and M. Sadoghi, "The Bedrock of BFT: A unified platform for BFT protocol design and implementation," *CoRR*, vol. abs/2205.04534. (2022), 2022. arXiv: 2205.04534.

[12] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media. (2018), 2018.

[13] V. Arora, T. Mittal, D. Agrawal, A. El Abbadi, X. Xue, *et al.*, "Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). (2017)*, 2017.

[14] G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi, "Proofs of Space: When space is of the essence," in *Security and Cryptography for Networks*, 538–557. (2014), 2014.

[15] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, "Achievable cases in an asynchronous environment," in *Annual Symposium on Foundations of Computer Science (SFCS)*, 337–346. (1987), 1987.

[16] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Communications of the ACM*, vol. 56, no. 5, 2013, 55–63. (2013).

[17] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Innovative Data Systems Research (CIDR)*, 223–234. (2011), 2011.

[18] I. Bentov, A. Gabizon, and A. Mizrahi, "Cryptocurrencies without proof of work," in *Financial Cryptography and Data Security*, 142–157. (2016), Springer, 2016.

[19] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld, "Proof of activity: Extending bitcoin's proof of work via proof of stake (extended abstract)," *SIGMETRICS Performance Evaluation Review*, vol. 42, no. 3, 2014, 34–37. (2014).

[20] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *ACM International Conference on Management of Data (SIGMOD)*, 1–10. (1995), 1995.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*, vol. 370. Addison-Wesley, 1987. URL: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx.

[22] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *USENIX Annual Technical Conference (ATC)*, 169–180. (2013), 2013.

[23] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 111–120. (2012), 2012.

[24] K. P. Birman and R. V. Renesse, *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press. (1993), 1993.

[25] BitShares-Core Contributors, *Bitshares documentation*, 2020. URL: https://how.bitshares.works/_/downloads/en/master/pdf/.

[26] M. Bravo, G. V. Chockler, and A. Gotsman, "Making byzantine consensus live," in *International Symposium on Distributed Computing (DISC)*, ser. LIPIcs, vol. 179, 23:1–23:17. (2020), 2020.

[27] M. Bravo, G. V. Chockler, and A. Gotsman, "Making byzantine consensus live," *Distributed Computing*, vol. 35, no. 6, 2022, 503–532. (2022).

[28] M. F. Bridgland and R. J. Watro, "Fault-tolerant decision making in totally asynchronous distributed systems," in *ACM Symposium on Principles of Distributed Computing*, 52–63. (1987), 1987.

[29] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph. (2016), 2016.

[30] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938. (2018), 2018. arXiv: 1807.04938.

[31] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 335–350. (2006), 2006.

[32] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437. (2017), 2017. arXiv: 1710.09437.

[33] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 173–186. (1999), 1999.

[34] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, 2002, 398–461. (2002).

[35] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 398–407. (2007), 2007.

[36] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM (JACM)*, vol. 43, no. 4, 1996, 685–722. (1996).

[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, 2008, 1–26. (2008).

[38] A. Charapko, A. Ailijiang, and M. Demirbas, "Pigpaxos: Devouring the communication bottlenecks in distributed consensus," in *ACM International Conference on Management of Data (SIGMOD)*, 235–247. (2021), 2021.

[39] J. Chen, S. Gupta, S. Rahnama, and M. Sadoghi, "Power-of-Collaboration: A sustainable resilient ledger built democratically," *IEEE Data Engineering Bulletin*, vol. 45, no. 2, 2022, 25–36. (2022).

[40] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013, 1–22. (2013).

[41] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient BFT consensus," in *ACM European Conference on Computer Systems (EuroSys)*, 34–50. (2022), 2022.

[42] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *ACM International Conference on Management of Data (SIGMOD)*, 123–140. (2019), 2019.

[43] D. Dolev and H. R. Strong, "Distributed commit with bounded waiting," in *IEEE Symposium on Reliability in Distributed Software and Database Systems*, 53–59. (1982), Jul. 1982.

[44] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM (JACM)*, vol. 34, no. 1, 1987, 77–97. (1987).

[45] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 33, no. 3, 1986, 499–516. (1986).

[46] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, 1988, 288–323. (1988).

[47] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, "Proofs of Space," in *Advances in Cryptology (CRYPTO)*, 585–605. (2015), 2015.

[48] R. Elmasri and S. B. Navathe, *Database systems*, vol. 9. Pearson Education. (2011), 2011.

[49] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, 1985, 374–382. (1985).

[50] H. Garcia-Molina and D. Barbara, "How to assign votes in a distributed system," *Journal of the ACM (JACM)*, vol. 32, no. 4, 1985, 841–860. (1985).

[51] S. Gazzaz, V. Chakraborty, and F. Nawab, "Croesus: Multi-stage processing and transactions for video-analytics in edge-cloud systems," in *IEEE International Conference on Data Engineering (ICDE)*, 1463–1476. (2022), 2022.

[52] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM Symposium on Operating Systems Principles (SOSP)*, 29–43. (2003), 2003.

[53] D. K. Gifford, "Weighted voting for replicated data," in *ACM Symposium on Operating Systems Principles (SOSP)*, 150–162. (1979), 1979.

[54] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 568–580. (2019), 2019.

[55] V. Gramoli, L. Bass, A. Fekete, and D. W. Sun, "Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, 2015, 2711–2724. (2015).

[56] J. N. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*, ser. Lecture Notes in Computer Science, vol. 60, Springer, 1978, 393–481. (1978).

[57] J. Gray, "The transaction concept: Virtues and limitations (invited paper)," in *International Conference on Very Large Data Bases (VLDB)*, 144–154. (1981), 1981.

[58] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, 2006, 133–160. (2006).

[59] R. Guerraoui, "Revisiting the relationship between non-blocking atomic commitment and consensus," in *International Workshop on Distributed Algorithms*, 87–100. (1995), 1995.

[60] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *ACM Symposium on Cloud Computing (SoCC)*, 1–16. (2016), 2016.

[61] S. Gupta, M. J. Amiri, and M. Sadoghi, "Chemistry behind agreement," in *Conference on Innovative Data Systems Research (CIDR). (2023)*, 2023.

[62] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "Proof-of-execution: Reaching consensus through fault-tolerant speculation," in *International Conference on Extending Database Technology (EDBT)*, 301–312. (2021), 2021.

[63] S. Gupta, J. Hellings, and M. Sadoghi, "Brief announcement: Revisiting consensus protocols through wait-free parallelization," in *International Symposium on Distributed Computing (DISC)*, vol. 146, 44:1–44:3. (2019), 2019.

[64] S. Gupta, J. Hellings, and M. Sadoghi, *Fault-Tolerant Distributed Transactions on Blockchain*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers. (2021), 2021.

[65] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: resilient concurrent consensus for high-throughput secure transaction processing," in *IEEE International Conference on Data Engineering (ICDE)*, 1392–1403. (2021), 2021.

[66] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, 2020, 868–883. (2020).

[67] S. Gupta, S. Rahnama, E. Linsenmayer, F. Nawab, and M. Sadoghi, "Reliable transactions in serverless-edge architecture," in *IEEE International Conference on Data Engineering (ICDE). (2023)*, 2023.

[68] S. Gupta, S. Rahnama, S. Pandey, N. Crooks, and M. Sadoghi, "Dissecting BFT consensus: In trusted components we trust!" In *ACM European Conference on Computer Systems (EuroSys)*, 2023.

[69] S. Gupta, S. Rahnama, and M. Sadoghi, "Permissioned blockchain through the looking glass: Architectural and implementation lessons learned," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 754–764, 2020.

[70] S. Gupta and M. Sadoghi, "EasyCommit: A non-blocking two-phase commit protocol," in *International Conference on Extending Database Technology (EDBT)*, 157–168. (2018), 2018.

[71] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University. (1994), Tech. Rep., 1994.

[72] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM computing surveys (CSUR)*, vol. 15, no. 4, 1983, 287–317. (1983).

[73] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *Conference on Innovative Data Systems Research (CIDR). (2019)*, 2019.

[74] J. Hellings, S. Gupta, S. Rahnama, and M. Sadoghi, "On the correctness of speculative consensus," *CoRR*, vol. abs/2204.03552. (2022), 2022. arXiv: 2204.03552.

[75] J. Hellings, D. P. Hughes, J. Primero, and M. Sadoghi, "Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing," *CoRR*, vol. abs/2008.04450. (2020), 2020. arXiv: 2008.04450.

[76] J. Hellings and M. Sadoghi, "Brief announcement: The fault-tolerant cluster-sending problem," in *International Symposium on Distributed Computing (DISC)*, 45:1–45:3. (2019), 2019.

[77] J. Hellings and M. Sadoghi, "Byshard: Sharding in a byzantine environment," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, 2021, 2230–2243. (2021).

[78] J. Hellings and M. Sadoghi, "Byzantine cluster-sending in expected constant communication," *CoRR*, vol. abs/2108.08541. (2021), 2021. arXiv: 2108.08541.

[79] J. Hellings and M. Sadoghi, "The fault-tolerant cluster-sending problem," in *International Symposium on Foundations of Information and Knowledge Systems (FoIKS)*, 168–186. (2022), 2022.

[80] M. Herlihy, L. Shrira, and B. Liskov, "Cross-chain deals and adversarial commerce," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, 2019, 100–113. (2019).

[81] H. Howard, A. Charapko, and R. Mortier, "Fast flexible paxos: Relaxing quorum intersection for fast paxos," in *International Conference on Distributed Computing and Networking*, 186–190. (2021), 2021.

[82] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible paxos: Quorum intersection revisited," in *International Conference on Principles of Distributed Systems (OPODIS)*, vol. 70, 25:1–25:14. (2016), 2017.

[83] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *USENIX Annual Technical Conference (ATC). (2010)*, 2010.

[84] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383. (2019), 2019. arXiv: 1902.03383.

[85] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 245–256. (2011), 2011.

[86] M. F. Kaashoek and A. S. Tanenbaum, "Group communication in the amoeba distributed operating system," in *International Conference on Distributed Computing Systems*, 222–230. (1991), 1991.

[87] M. Kazhamiaka, B. Memon, C. Kankanamge, S. Sahu, S. Rizvi, B. Wong, and K. Daudjee, "Sift: Resource-efficient consensus with rdma," in *International Conference on Emerging Networking Experiments And Technologies*, 260–271. (2019), 2019.

[88] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *ACM Symposium on Principles of Distributed Computing (PODC)*, A. Miller, K. Censor-Hillel, and J. H. Korhonen, Eds., 165–175. (2021), 2021.

[89] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez, "Database replication," *Synthesis Lectures on Data Management*, vol. 5, no. 1, 2010, pp. 1–153.

[90] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure Proof-of-Stake blockchain protocol," in *Advances in Cryptology (CRYPTO)*, 357–388. (2017), Springer, 2017.

[91] S. King and S. Nadal, *PPCoin: Peer-to-peer crypto-currency with Proof-of-Stake. (2012)*, 2012. URL: https://www.peercoin.net/whitepapers/peercoin-paper.pdf.

[92] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A secure, scale-out, decentralized ledger via sharding," in *IEEE Symposium on Security and Privacy (S&P)*, 583–598. (2018), 2018.

[93] J. Kończak, P. T. Wojciechowski, N. Santos, T. Żurkowski, and A. Schiper, "Recovery algorithms for paxos-based state machine replication," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, 2019, 623–640. (2019).

[94] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computing Systems*, vol. 27, no. 4, 2009, 7:1–7:39. (2009).

[95] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *ACM European Conference on Computer Systems (EuroSys)*, 113–126. (2013), 2013.

[96] P. Kuznetsov, A. Tonkikh, and Y. X. Zhang, "Revisiting optimal resilience of fast byzantine consensus," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 343–353. (2021), 2021.

[97] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, no. 7, 1978, 558–565. (1978).

[98] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, 1998, 133–169. (1998).

[99] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, 2001, 18–25. (2001).

[100] L. Lamport, "Generalized consensus and paxos," *Technical Report MSR-TR-2005-33, Microsoft Research. (2005)*, 2005.

[101] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, 2006, 79–103. (2006).

[102] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable paxos," *TechReport, Microsoft Research. (2008)*, 2008.

[103] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *ACM symposium on Principles of Distributed Computing (PODC)*, 312–313. (2009), 2009.

[104] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, 2010, 63–73. (2010).

[105] L. Lamport and M. Massa, "Cheap paxos," in *International Conference on Dependable Systems and Networks*, 307–314. (2004), 2004.

[106] L. Lamport, R. Shortak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, 1982, 382–401. (1982).

[107] B. Lampson and D. Lomet, "A new presumed commit optimization for two phase commit," in *International Conference on Very Large Data Bases (VLDB)*, 630–640. (1993), 1993.

[108] B. Lampson and H. E. Sturgis, "Crash recovery in a distributed data storage system," in *Computer Science Lab, Xerox Parc, Palo Alto, CA, Technical Report. (1976)*, 1976.

[109] B. W. Lampson, "How to build a highly available system using consensus," in *International Workshop on Distributed Algorithms*, 1–17. (1996), 1996.

[110] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to paxos overhead: Replacing consensus with network ordering," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 467–483. (2016), 2016.

[111] W. Li, S. Andreina, J.-M. Bohli, and G. Karame, "Securing Proof-of-Stake blockchain protocols," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 297–315. (2017), Springer, 2017.

[112] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," in *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 103–115. (2006), 2006.

[113] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *ACM SIGSAC Conference on Computer and Communications Security*, 17–30. (2016), 2016.

[114] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, 2013, 661–672. (2013).

[115] S. Maiyya, F. Nawab, D. Agrawal, and A. E. Abbadi, "Unifying consensus and atomic commitment for effective cloud data management," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, 2019, 611–623. (2019).

[116] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machine for wans," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 369–384. (2008), 2008.

[117] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 527–536. (2010), 2010.

[118] M. Mihai Letia, N. Preguica, and M. Shapiro, "Crdts: Consistency without concurrency control," *RR-6956, INRIA. (2009)*, 2009.

[119] N. Mittal and F. Nawab, "Coolsm: Distributed and cooperative indexing across edge and cloud machines," in *IEEE International Conference on Data Engineering (ICDE)*, 420–431. (2021), 2021.

[120] C. Mohan and B. Lindsay, "Efficient commit protocols for the tree of processes model of distributed transactions," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 2, 1985, 40–52. (1985).

[121] C. Mohan, R. Strong, and S. Finkelstein, "Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors," in *Proceedings of the annual ACM symposium on Principles of distributed computing (PODC)*, 89–103 (1983), 1983.

[122] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *ACM Symposium on Operating Systems Principles (SOSP)*, 358–372. (2013), 2013.

[123] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 517–532. (2016), 2016.

[124] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2009. URL: https://bitcoin.org/bitcoin.pdf.

[125] F. Nawab, "Wedgechain: A trusted edge-cloud store with asynchronous (lazy) trust," in *IEEE International Conference on Data Engineering (ICDE)*, 408–419. (2021), 2021.

[126] F. Nawab, D. Agrawal, and A. El Abbadi, "Dpaxos: Managing data closer to users for low-latency and mobile applications," in *ACM International Conference on Management of Data (SIGMOD)*, 1221–1236. (2018), 2018.

[127] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 8–17. (1988), 1988.

[128] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference (ATC)*, 305–319. (2014), 2014.

[129] M. T. Ozsu and P. Valduriez, *Principles of distributed database systems*, vol. 2. Springer. (1999), 1999.

[130] S. Park, A. Kwon, G. Fuchsbauer, P. Gaži, J. Alwen, and K. Pietrzak, "SpaceMint: A cryptocurrency based on proofs of space," in *Financial Cryptography and Data Security*, 480–499. (2018), Springer, 2018.

[131] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. E. Abbadi, "Serializability, not serial: Concurrency control and availability in multi-datacenter datastores," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, 2012, 1459–1470. (2012).

[132] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, 1980, 228–234. (1980).

[133] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," in *European conference on parallel processing (Euro-Par)*, 513–520. (1998), 1998.

[134] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases (DAPD)*, vol. 14, no. 1, 2003, 71–98. (2003).

[135] S. Rahnama, S. Gupta, R. Sogani, D. Krishnan, and M. Sadoghi, "Ringbft: Resilient consensus over sharded ring topology," in *International Conference on Extending Database Technology (EDBT)*, 2:298–2:311. (2022), 2022.

[136] R. Ramakrishnan, J. Gehrke, and J. Gehrke, *Database management systems*, vol. 3. McGraw-Hill New York. (2003), 2003.

[137] M. Sadoghi and S. Blanas, *Transaction Processing on Modern Hardware*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers. (2019), 2019.

[138] N. Santos and A. Schiper, "Optimizing paxos with batching and pipelining," *Theoretical Computer Science*, vol. 496, 2013, 170–183. (2013).

[139] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, 1990, 299–319. (1990).

[140] A. Sharov and A. S. A. M. M. Stokely, "Take me to your leader! online optimization of distributed storage configurations," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, 2015, 1490–1501. (2015).

[141] D. Skeen, "Nonblocking commit protocols," in *ACM International Conference on Management of Data (SIGMOD)*, 133–142. (1981), 1981.

[142] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Transactions on Software Engineering*, no. 3, 1983, 219–228. (1983).

[143] J. Sousa and A. Bessani, "Separating the wheat from the chaff: An empirical design for geo-replicated state machines," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 146–155. (2015), 2015.

[144] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2705–2718. (2022), 2022.

[145] I. Stanoi, D. Agrawal, and A. El Abbadi, "Using broadcast primitives in replicated databases," in *International Conference on Distributed Computing Systems (ICDCS)*, 148–155 (1998), 1998.

[146] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, 1979, 180–209. (1979).

[147] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *ACM International Conference on Management of Data (SIGMOD)*, 1–12. (2012), 2012.

[148] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, 2015, 42:1–42:36. (2015).

[149] R. Van Renesse, N. Schiper, and F. B. Schneider, "Vive la différence: Paxos vs. viewstamped replication vs. zab," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 4, 2014, 472–484. (2014).

[150] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 135–144. (2009), 2009.

[151] M. Whittaker, "Compartmentalizing state machine replication," Ph.D. dissertation, University of California, Berkeley. (2021), 2021.

[152] M. Whittaker, N. Giridharan, A. Szekeres, J. Hellerstein, H. Howard, F. Nawab, and I. Stoica, "Matchmaker Paxos: A Reconfigurable Consensus Protocol," *Journal of Systems Research (JSys)*, vol. 1, no. 1, Sep. 2021.

[153] M. J. Whittaker, N. Giridharan, A. Szekeres, J. M. Hellerstein, H. Howard, F. Nawab, and I. Stoica, "Matchmaker paxos: A reconfigurable consensus protocol [technical report]," *CoRR*, vol. abs/2007.09468. (2020), 2020. arXiv: 2007.09468.

[154] M. J. Whittaker, N. Giridharan, A. Szekeres, J. M. Hellerstein, and I. Stoica, "Bipartisan paxos: A modular state machine replication protocol," *CoRR*, vol. abs/2003.00331. (2020), 2020. arXiv: 2003.00331.

[155] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, 2016. URL: https://gavwood.com/paper.pdf.

[156] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *ACM Symposium on Operating Systems Principles (SOSP)*, 292–308. (2013), 2013.

[157] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 347–356. (2019), 2019.

[158] V. Zakhary, D. Agrawal, and A. E. Abbadi, "Atomic commitment across blockchains," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, 2020, 1319–1331. (2020).

[159] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi, "Global-scale placement of transactional data stores.," in *International Conference on Extending Database Technology (EDBT)*, 385–396. (2018), 2018.

[160] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi, "Db-risk: The game of global database placement," in *International Conference on Management of Data (SIGMOD)*, 2185–2188. (2016), 2016.

[161] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 931–948. (2018), 2018.

[162] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports, "Building consistent transactions with inconsistent replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, 2018, 1–37. (2018).